Courant Computer Science Report #18

September 1979

LEVEL

Automatic Discovery of Heuristics for Nondeterministic Programs from Sample Execution Traces

Salvatore J. Stolfo

D D C
RECEIVED
NOV 15 1979
E

Courant Institute of
Mathematical Sciences

Computer Science Department

New York University

79 11 13 233

COURANT COMPUTER SCIENCE PUBLICATIONS

COURANT COMPUTER SCIENCE NOTES

A101 ABRAHAMS, P. The PL/I Programming Language, 1979, 151 p.

C66 COCKE, J. & SCHWARTZ, J. Programming Languages and Their Compilers, 1970, 767 p.

D86 DAVIS, M. Computability, 1974, 248 p.

M72 MANACHER, G. ESPL: A Low-Level Language in the Style of Algol, 1971, 496 p.

M81 MULLISH, H. & GOLDSTEIN, M. A SETLB Primer, 1973, 201 p.

S91 SCHWARTZ, J. On Programming: An Interim Report on the SETL Project. Generalities; The SETL Language and Examples of Its Use. 1975, 675 p.

S99 SHAW, P. GYVE — A Programming Language for Protection and Control in a Concurrent Processing Environment, 1978, 668 p.

S100 SHAW, P. " Vol. 2, 1979, 600 p.

W78 WHITEHEAD, E.G., Jr. Combinatorial Algorithms, 1973, 104 p.

COURANT COMPUTER SCIENCE REPORTS

1 WARREN, H. Jr. ASL: A Proposed Variant of SETL, 1973, 326 p.

2 HOBBS, J. R. A Metalanguage for Expressing Grammatical Restrictions in Nodal Spans Parsing of Natural Language, 1974, 266 p.

3 TENENBAUM, A. Type Determination for Very High Level Languages, 1974, 171 p.

5 GEWIRTZ, W. Investigations in the Theory of Descriptive Complexity, 1974, 60 p.

6 MARKSTEIN, P. Operating System Specification Using Very High Level Dictions, 1975, 152 p.

7 GRISHMAN, R. (ed.) Directions in Artificial Intelligence: Natural Language Processing, 1975, 107 p.

8 GRISHMAN, R. A Survey of Syntactic Analysis Procedures for Natural Language, 1975, 94 p.

9 WEIMAN, CARL Scene Analysis: A Survey, 1975, 62 p.

10 RUBIN, N. A Hierarchical Technique for Mechanical Theorem Proving and Its Application to Programming Language Design, 1975, 172 p.

11 HOBBS, J.R. & ROSENSCHEIN, S.J. Making Computational Sense of Montague's Intensional Logic, 1977, 41 p.

12 DAVIS, M. & SCHWARTZ, J. Correct-Program Technology/Extensibility of Verifiers, with an Appendix by E. Deak, 1977, 146 p.

13 SEMENIUK, C. Groups with Solvable Word Problems, 1979, 77 p.

14 FABRI, J. Automatic Storage Optimization, 1979, 159 p. ..

15. LIU, S-C. & PAIGE, R. Data Structure Choice/Formal Differentiation. Two Papers on Very High Level Program Optimization, 1979, 658 p.

16 GOLDBERG, A. T. On the Complexity of the Satisfiability Problem, 1979, 85 p.

17 SCHWARTZ, J.T. & SHARIR, M. A Design for Optimizations of the Bitvectoring Class, 1979, 71 p.

Notes: Available from Department LN. Prices on request.
Reports: Available from Ms. Lenora Greene. Nos. 1,3, 6,7,8,10 available in xerox only..

COURANT INSTITUTE OF MATHEMATICAL SCIENCES

Sep 79

NOV 15 1979

Computer Science   NSO-18

Automatic Discovery of Heuristics for Nondeterministic

Programs from Sample Execution Traces

Salvatore J. Stolfo

ii

Acknowledgements

I am indebted to Malcolm Harrison for providing me
with guidance and support throughout this research. I
would like to express my thanks to my colleagues Allen
Goldberg and Sandra Rapps, my wife Joanne Stolfo, my
parents Sam and Rose Stolfo, and the excellent typists
Connie Engle, Jeanette Figueroa and Lisa Walsh.

Table of Contents

CHAPTER 1. INTRODUCTION

1.1 Background and Scope of Thesis

During the last few years, a number of relatively effective artificial intelligence (AI) programs have been written incorporating considerable amounts of knowledge. Consequently, the problem of encoding such knowledge in a useful form has emerged as one of the central problems of AI. Winograd [48] distinguishes between *declarative* information which can be thought of as "knowing what", and *procedural* information which can be thought of as "knowing how". He describes the underlying problem as that of constructing representations which can take advantage of the decomposability of the declarative form without sacrificing the interactive possibilities of the procedural form. Ideally, it should be possible to specify information in a form which does not constrain the way in which it is to be used. Unfortunately, a straightforward implementation of such a declarative representation corresponds to a nondeterministic program which makes a relatively blind search through the solution space.

More recently, attention has turned towards mechanisms which facilitate incorporating limited procedural or heuristic information into a primarily declarative framework. Examples

include Sickel's Clause Interconnectivity Graphs [39] in a
resolution theorem-proving framework, Robot Plans in the
STRIPS problem-solving system [14], and procedural deduc-
tions in semantic networks [10]. In [36], Rychener discusses
the approach of building a rational "goal" structure into
declarative rule-based (production) systems, while Davis [9]
favors a separate set of "meta-rules" specifying control
information in a hierarchical fashion. In some cases we
might expect that the amount of procedural information is of
the same order of magnitude as the amount of declarative
information, and so such approaches may be effective. However,
in most cases the amount of control information will be very
large (such as theorem-proving, natural language
understanding, vision).   Thus the problem of acquiring,
debugging and extending control information will become
increasingly important.

In general, we might expect that this control informa-
tion will embody very sophisticated principles requiring
data structures not present in the declarative form of the
program, or which are deducible only by the use of consider-
able intelligence. For example, it is clear that the
 deduction of the heuristic principles of evaluation and
$\alpha-\beta$ search from a declarative chess program which specifies
only the rules of chess without any strategies for playing
will require an analysis which is considerably beyond the
ability of present techniques; even the optimization of the

- 2 -

parameters in a linear evaluation function involves highly
sophisticated processing.

However, it is the contention of this thesis that
there are a number of important areas in which it might be
possible to deduce control information automatically from
a declarative program. These include declaratively speci-
fied problems:

- for which there exists a relatively simple algorithmic
  procedure
- whose performance can be improved in frequently occur-
  ring or particularly important special cases
- in which particular subproblems can be solved by
  simple algorithmic procedures.

In this thesis, we will investigate a technique for improving
the performance of a nondeterministic program which is
based on an analysis of the behavior of the program on
sample inputs.

The following section presents a sample problem used
throughout this document to demonstrate the proposed
techniques. In Section 1.3, the approach we take to this
problem is outlined in detail along with the technical
problems involved in the actual analysis. The final section
of this chapter presents a history of the research related
to this problem and which supports the approach we use.

- 3 -

1.2 A Sample Problem: Jigsaw Puzzles

As an example, consider the following problem. Suppose
we want to solve a (slightly idealized) jigsaw puzzle in
which each piece has an average color and four sides
described by unique integers (with side i fitting side -i).
We consider below a nondeterministic algorithm written
in *Production System* (PS) form [8,32,37].

A PS is a (nondeterministic) program consisting of a
set of *productions* or *rules*, called *Production Memory*
together with a data base of assertions, called the
Working Memory (WM) set. Each rule consists of a conjunc-
tion of patterns of data elements, called the *Left-hand Side*
(LHS) and a series of actions called the *Right-hand Side*
(RHS). The RHS specifies information that is to be
deposited in or removed from the WM. Execution of the
program consists of iterating the following sequence of
actions (each iteration is called a *cycle*):
(1) for each rule, determine if its LHS matches the current
     environment in WM   (multiple instantiations are possible)
(2) from the set of rules satisfying step (1), called the
     Conflict Set (CS), nondeterministically select one.
(3) *fire* the rule selected in step (2), that is, apply
     the actions specified in the RHS.

In the exact form of the PS representation we use,
data elements can be any LISP data structure. An atomic
data element in the LHS of a production must match an exact

- 4 -

data element in WM   and a list must match a list with the
same structure and content. A symbol preceded with an
equals sign (=) represents a variable which can match any
data structure. The $ symbol contained in the productions
has the same function as the SNOBOL4 immediate assignment
operator. When a rule is fired, the matching data elements
are not deleted from WM unless they are included as argu-
ments to the <delete> system function in the RHS of the rule.
The other system functions are represented in lower case and
enclosed in pointed brackets (< >). Their function is
described by their names. The operator - in the LHS has
the same function as <not>. The symbol ! is an operator
which matches the entire remaining portion of the list that
contains it. Where it appears in the RHS it deposits the
list that matched it but without the enclosing parentheses.
The complete program is given in the following sections.

1.2.1   A Jigsaw Puzzle Production System

The representation we have chosen models the puzzle-
solver as having an eye with which to focus on a single
object, a hand with which to grasp an object and a limited
memory which can remember a single object, a single pile,
and a particular color at any one time. The puzzle is
presented to the puzzle-solver as a heap (or unordered set)
of puzzle pieces. The productions allow for the piling of
objects, and through the functions of the eye, hand and
memory, the systematic scanning of an ordered pile of puzzle
pieces. Various sensing productions (those without actions
in the RHS) indicate a variety of conditions of WM as is
outlined below.

Working Memory Data Elements:

(PIECE p c)   represents a puzzle piece where p is a unique
integer associated with a piece and c is its average color.
We will assume the machine knows at any time where piece p
is located and so we ignore location in the representation.

(s p n c)   represents the side of a puzzle piece. s is
either L, R, T or B, representing the left, right, top and
bottom, respectively, of piece p. Since location is ignored,
rotation of a piece in space is also ignored. n is an
integer representing the shape of the side where +n mates
with -n and 0 represents a straight edge. c is the color

- 6 -

of the edge. The representation could easily be extended
to represent a vector of colors for a series of points on
the edge.

(NUMBER-OF-PIECES n) states that n is the total number of
puzzle pieces.

(NUMBER-IN-HEAP n) states that n is the total number of
pieces in the heap.

(IN-PUZZLE (PIECE p c)) states that piece p is in the
puzzle.

(NUMBER-IN-PUZZLE n) states that n is the total number of
pieces in the puzzle.

(JOINED (side$_1$ side$_2$) (side$_3$ side$_4$) ...) represents the
sequence of sides that were joined in forming the puzzle.
The symbol side$_i$ is of the form (s p n c) above.

(BEING-PUT-IN-PUZZLE x) represents the current piece x that
is actively being placed in the puzzle.

(IN-HEAP (PIECE p c)) represents puzzle piece p as being
in a heap with no implicit ordering of edges.

(LOOKING-AT x) states that object x is in view. If nothing
is in view, then (LOOKING-AT NOTHING) is in WM.

(HOLDING x) states that x is the current object being held.
If the hand is empty, (HOLDING NOTHING) is in WM.
(CURRENT-COLOR c) represents the current color being considered.

- 7 -

Production Memory:

LOOK-AT-PIECE-IN-HEAP:

    If a piece is in the heap, then you can look at it.

        [(LOOKING-AT =anything) $ =c1
         (IN-HEAP (PIECE =p =c) $ =object)
         -(LOOKING-AT =object)
                                    -->
                                        (<delete> =c1)
                                        (LOOKING-AT =object)]

LOOK-AT-OBJECT-IN-HAND:

    If something is in your hand, then you can look at it.

        [(LOOKING-AT =something) $ =c1
         (HOLDING =object $ (<not> NOTHING))
         -(HOLDING =something)
                                    -->
                                        (<delete> =c1)
                                        (LOOKING-AT =object)]

CLOSE-EYES:

    If you are looking at something which is not NOTHING, then you can

look away from it.

        [(LOOKING-AT =something) $ =c1
         -(LOOKING-AT NOTHING)
                                    -->
                                        (<delete> =c1)
                                        (LOOKING-AT NOTHING)]

OBJECT-IN-HAND-IN-VIEW:

    This production senses when the object being held is in view without

performing any actions.

        [(HOLDING =object)
         (LOOKING-AT =object)
         -(LOOKING-AT NOTHING)
                                    -->
                                        ]

PICK-UP-OBJECT-IN-VIEW:

    If you are looking at an object and holding NOTHING, then you can pick

up the object. Its position changes to being in your hand.

        [(HOLDING NOTHING) $ =c1
         (LOOKING-AT =object)
         (IN-HEAP =object) $ =c3
         (NUMBER-IN-HEAP =numb) $ =c4
                                    -->
                                        (<delete> =c1 =c3 =c4)
                                        (<NUMBER-IN-HEAP (<sub1> =numb))
                                        (LOOKING-AT =object)
                                        (HOLDING =object)]

PUT-PIECE-DOWN-IN-HEAP:

If you are holding a puzzle piece, you can put it in the heap and your
hand is empty. Notice that the piece continues to be in view.

```
[(HOLDING (PIECE =p =c)) $ =c1
 (LOOKING-AT =object)
 (NUMBER-IN-HEAP =numb) $ =c3
                            -->
                                (<delete> =c1 =c3)
                                (LOOKING-AT =object)
                                (HOLDING NOTHING)
                                (NUMBER-IN-HEAP (<add1> =numb))
                                (IN-HEAP (PIECE =p =c))]
```

FIND-COLOR-OF-PIECE:

If you are holding a puzzle piece, you can look at it and know its
color.

```
[(LOOKING-AT (PIECE =p =c))
 (CURRENT-COLOR =anything) $ =c2
                            -->
                                (<delete> =c2)
                                (CURRENT-COLOR =c)]
```

FORGET-COLOR-OF-PIECE:

If a Piece is in view and you know its color, you can look away from it
and forget its color.

```
[(LOOKING-AT (PIECE =p =c)) $ =c1
 (CURRENT-COLOR =c) $ =c2
                            -->
                                (<delete> =c1 =c2)
                                (LOOKING-AT NOTHING)
                                (CURRENT-COLOR NOTHING)]
```

PIECE-HAS-STRAIGHT-EDGE:

This production senses when the piece in view has a straight edge.

```
[(LOOKING-AT (PIECE =p =c))
 (=any =p 0 =c)
                            -->
                                ]
```

PIECE-HAS-CURRENT-COLOR:

This senses when the object in view has the current color you are
considering.

```
[(LOOKING-AT (PIECE =p =c))
 (CURRENT-COLOR =c)
                            -->
                                ]
```

EMPTY-HEAP:

This senses if the heap is empty.

```
[-(IN-HEAP =anything)
  (NUMBER-IN-HEAP 0) $ =c2
                                  -->
                                     (<delete> =c2)]
```

START-PUZZLE:

If there is no puzzle currently being built and you are holding a piece, then you can make this piece the first part of the puzzle, leaving your hand empty. Notice that you continue to look at the piece.

```
[-(IN-PUZZLE =anything)
  (HOLDING (PIECE =p =c) $ =object) $ =c2
  (LOOKING-AT =object)
                            -->(JOINED )
                               (<delete> =c1 =c2)
                               (HOLDING NOTHING)
                               (IN-PUZZLE =object)
                               (LOOKING-AT =object)
                               (NUMBER-IN-PUZZLE 1)]
```

PIECE-FITS-IN-PUZZLE:

This production senses when a puzzle piece that is in view will fit another puzzle piece already in the puzzle.

```
[(LOOKING-AT (PIECE =p =c) $ =object)
 -(IN-PUZZLE =object)
  (IN-PUZZLE (PIECE =p2 =otherc) $ =otherp)
  (=any =p =n =anyc)
  (=another =p2 (<negative> =n) =k)
                            --> ]
```

FIT-PIECE-IN-PUZZLE:

If the piece being held and in view fits in the puzzle, the sides of the matching pieces can be joined.

```
[(HOLDING (PIECE =p =c) $ =object)
 -(IN-PUZZLE =object)
  (LOOKING-AT =object)
  (IN-PUZZLE (PIECE =p2 =otherc) $ =otherp)
  (<forall> (=any =p =n =c) $ =c4
            (=another =p2 (<negative> =n) =k) $ =c5 )
  (JOINED ! =rest) $ =c6
                            -->
                               (<delete> (<all> =c4 =c5) =c6)
                               (BEING-PUT-IN-PUZZLE =object)
                               (LOOKING-AT =object)
                               (JOINED ! =rest (<all> =c4 =c5))]
```

- 10 -

PIECE-PUT-IN-PUZZLE:

     If you are holding a puzzle piece, which is currently being placed in

the puzzle, you can put it down in the puzzle.

```
[(HOLDING (PIECE =p =c) $ =object) $ =c1
 (NUMBER-IN-PUZZLE =m) $ =c2
-(IN-PUZZLE =object)
 (BEING-PUT-IN-PUZZLE =object) $ =c4
                            -->
                                (<delete> =c1 =c2 =c4)
                                (HOLDING NOTHING)
                                (LOOKING-AT NOTHING)
                                (IN-PUZZLE =object)
                                (NUMBER-IN-PUZZLE (<add1> =m))]
```

PUZZLE-IS-FINISHED:

     If all of the pieces are in the puzzle, you can stop.

```
[(NUMBER-IN-PUZZLE =n)
 (NUMBER-OF-PIECES =n)
-(IN-HEAP =anything)
                    -->
                        (<halt>)]
```

- 11 -

It should be noted that this representation is highly nondeterministic and if executed would be impossibly inefficient (even in the presence of any "syntactically-oriented" selection rules [30], as nearly 10 years of resolution theorem-proving refinement research demonstrates). It does not even "know" that it will find a solution (i.e. terminate) if it can repeat the production **FIT**-PIECE-IN-PUZZLE as frequently as possible, so even the crudest goal-subgoal structure is absent. (However, there is no production to remove a piece from the puzzle, which could serve as a clue to an intelligent observer.) It is even possible for the program to pick up a piece and then put it down immediately without doing anything with it.

The performance of this program could be improved to a tolerable level if the following very simple heuristic were added:

- use the following sequence repeatedly:

  - pick up a piece from the heap;

  - insert it in the puzzle if it fits;

  - if not, put it back in the heap.

An alternative heuristic would be:

- use the following sequence repeatedly:

  - pick a piece p in the puzzle with a missing neighbor;

  - pick a piece from the heap;

  - if it matches a side of p, insert it in the puzzle;

  - if not, place it back in the heap.

- 12 -

Each of these heuristics is very simple, consisting mainly of sequencing rules for the productions, and would seem to be within the range of automatic inference. A further improvement could be made by noting that the procedure for selecting a piece from the heap is inefficient since the same piece may be selected repeatedly (and in a production system such as OPS2 [15] which prefers to use recently referenced items in WM, in other respects a reasonable strategy, the search will almost always be ineffective). If we add a few more productions, it becomes possible to search systematically through the heap by constructing an ordered pile from the pieces in the heap, and putting a piece down in a different pile when it has looked at it. A pile is more precisely a queue, but we prefer to be consistent with puzzle terminology. The actual pile productions are given in the next section.

1.2.2   Extensions to Jigsaw Puzzle PS

Working Memory Data Elements:

(NUMBER-OF-PILES m)   m is the number of piles we have
made (initially zero).

(CURRENT-PILE n)   represents the current pile we are using.
Each pile created is assigned a unique integer, n.

(ALL-PILES p_1 p_2 ...)   represents all of the piles we have
made where p_i is the integer associated with a pile.

(PILE i t_1 t_2 ...)   represents the contents of pile i
where t_j can be any object, and t_1 is the first object
in the pile.

(REMEMBERED-OBJECT t)   represents object t as being tagged
or remembered as special.

(REMEMBERED-PILE i)   represents pile i as being special.

- 14 -

Production Memory:

MAKE-A-PILE:

If you want to make a pile and you are holding something, create a pile
with it and your hand is empty. The object continues to be in view.

```
[(HOLDING (<not> NOTHING) $ =object) $ =c1
 (LOOKING-AT =object)
 (NUMBER-OF-PILES =m) $ =c3
 (ALL-PILES ! =r) $ =c4
 (CURRENT-PILE NONE) $ =c5
                            -->
                                (<delete> =c1 =c3 =c4 =c5)
                                (NUMBER-OF-PILES (<add1> =m))
                                (LOOKING-AT =object)
                                (ALL-PILES (<add1> =m) ! =r)
                                (CURRENT-PILE (<add1> =m))
                                (PILE (<add1> =m) =object)
                                (HOLDING NOTHING)]
```

PICK-A-PILE:

If you are not working with a pile, just pick the first one.

```
[(CURRENT-PILE NONE) $ =c1
 (ALL-PILES =pl ! =r)
                            -->
                                (<delete> =c1)
                                (ALL-PILES =pl ! =r)
                                (CURRENT-PILE =pl)]
```

PICK-OBJECT-FROM-PILE:

To pick an object from a pile, if your hand is empty, reach in and
pick up the first one you see.

```
[(CURRENT-PILE =pl)
 (HOLDING NOTHING) $=c2
 (ALL-PILES =pl ! =r)
 (PILE =pl =object ! =rest) $ =c4
 (LOOKING-AT =object)
                            -->
                                (<delete> =c2 =c4)
                                (LOOKING-AT =object)
                                (HOLDING =object)
                                (PILE =pl ! =rest)]
```

PUT-OBJECT-IN-PILE:

To put an object in a pile, if you are holding something, just place
it in the back of the pile and your hand is empty.

```
[(CURRENT-PILE =pl)
 (HOLDING (<not> NOTHING) $ =object) $ =c2
 (LOOKING-AT =object)
 (ALL-PILES =pl ! =r)
 (PILE =pl ! =rest) $ =c5
                            -->
                                (<delete> =c2 =c5)
                                (LOOKING-AT =object)
                                (HOLDING NOTHING)
                                (PILE =pl ! =rest =object)]
```

- 15 -

FORGET-THIS-PILE:

    To forget a pile, just push the current pile behind all the others.

        [(CURRENT-PILE =p1) $ =c1
         (ALL-PILES =p1 ! =rest) $ =c2
                                    -->
                                        (<delete> =c1 =c2)
                                        (CURRENT-PILE NONE)
                                        (ALL-PILES ! =rest =p1)

PILE-IS-EMPTY:

    A pile is empty if there is nothing in it.

        [(CURRENT-PILE =p1)
         (ALL-PILES =p1 ! =rest)
         (PILE =p1)
                                    -->
                                        ]

DESTROY-A-PILE:

    To destroy the current pile, just strike it from memory.

        [(CURRENT-PILE =p1) $ =c1
         (ALL-PILES =p1 ! =rest) $ =c2
         (PILE =p1 ! =r) $ =c3
                                    -->
                                        (<delete> =c1 =c2 =c3)
                                        (CURRENT-PILE NONE)
                                        (ALL-PILES ! =rest)]

THERE-ARE-NO-PILES:

    There are no piles if all the piles were destroyed or none were ever
made.

        [(CURRENT-PILE NONE)
         (ALL-PILES)
                                    -->
                                        ]

LOOK-AT-FIRST-IN-PILE:

    If there is an object at the front of the pile, you can focus on it.

        [(LOOKING-AT =anything) $ =c1
         (CURRENT-PILE =p1)
         (PILE =p1 =object $ (<not> =anything)! =rest)
                                    -->
                                        (<delete> =c1)
                                        (LOOKING-AT =object)]

- 16 -

LOOK-AT-NEXT-IN-PILE:

If you want to scan through a pile and you are looking at the first object in it, focus on the next one by placing the first object behing the last one in the pile.

```
[(LOOKING-AT =object) $ =c1
 (CURRENT-PILE =p1)
 (PILE =p1 =object =next ! =r) $ =c3
                            -->
                                (<delete> =c1 =c3)
                                (LOOKING-AT =next)
                                (PILE =p1 =next ! =r =object)]
```

REMEMBER-CURRENT-PILE:

Whichever pile is current, tag it as special.

```
[(CURRENT-PILE =p1 $ (<not> NONE))
 (REMEMBERED-PILE =any) # =c2
                            -->
                                (<delete> =c2)
                                (REMEMBERED-PILE =p1)]
```

REMEMBERED-PILE-IS-CURRENT:

This senses if the current pile is the tagged special pile.

```
[(CURRENT-PILE =p1 $ (<not> NONE))
 (REMEMBERED-PILE =p1)
                            -->
                                ]
```

FORGET-REMEMBERED-PILE:

The special pile can always be forgotten.

```
[(REMEMBERED-PILE (<not> NONE)) $ =c1
                            -->
                                (<delete> =c1)
                                (REMEMBERED-PILE NONE)]
```

REMEMBER-CURRENT-OBJECT:

As for piles, objects can also be tagged as special, when they are in view.

```
[(LOOKING-AT =object $ (<not> NOTHING))
 (REMEMBERED-OBJECT =any) $ =c2
                            -->
                                (<delete> =c2)
                                (REMEMBERED-OBJECT =object)]
```

FORGET-REMEMBERED-OBJECT:

Same as for piles.

```
[(REMEMBERED-OBJECT (<not> NONE)) $ =cl
                                -->
                                    (<delete> =cl)
                                    (REMEMBERED-OBJECT NONE)]
```

REMEMBERED-OBJECT-IN-VIEW:

This production senses when the special object is in view.

```
[(REMEMBERED-OBJECT =object)
 (LOOKING-AT =object)
                                -->
                                ]
```

REMEMBERED-OBJECT-IN-HAND:

This senses when the special object is being held.

```
[(REMEMBERED-OBJECT =object)
 (HOLDING =object)
                                -->
                                ]
```

- 18 -

The pile productions are very general and might be expected to be present in any system. Indeed, it seems likely that productions for dealing with heaps and piles (i.e. sets and queues) and heuristics for implementing important operations such as searching will be present in production systems as they are in modern programming languages [11].

Carrying our jigsaw puzzle problem further, sufficient productions are present now to permit simple sequencing heuristics to reflect the usual strategies used by experienced puzzle-solvers:

- work on the outside edges first (build a pile of outside edges);

- repeat the **FIT**-PIECE-IN-PUZZLE sequence until this pile is empty;

- separate the pieces in the heap into piles of the same average color and search the appropriate pile first;

- search first for pieces which have more than one neighbor of the same average color (work on the sky first);

It may appear at first glance that automatic detection of such heuristics is a task of great difficulty. Below we outline an approach to this problem which has been quite successful.

1.3 Approach

Our approach is as follows:

(1) Select a 'typical' input to the program and run the
     program repeatedly on this input.

(2) Record the sequence of rules selected (together with
     input/output information and the conflict set of rules
     on each cycle).

(3) Repeat this for other typical inputs.

(4) Describe the better (i.e. shorter) successful sequences
     in a language, CRAPS, designed for this purpose (and
     described in the   next chapter).

(5) Generate a set of meta-rules whose objective is to aid
     the CRAPS description if the sequencing is inappropriate.

(6) Use the CRAPS description and the meta-rules to guide
     the program's subsequent decisions.

In view of the complexities of the problems with this
approach, it was necessary to define a starting point from
which we can proceed to study the more general case.
Accordingly, we have made certain assumptions and decisions
in order to derive useful results.

Inherent in our approach is the assumption that good
decision-making procedures or heuristics can be inferred
from the performance of the program on cnly selected inputs.
We anticipated that the selection of inputs would be
critical and that eventually new inputs would be handled
incrementally, as was done by Winston [49]. However, in this

work we concentrate on the simpler problem of getting a good solution for the nonincremental case. In general, as suggested by   work on learning systems [49] and information theory [6,26,40], we give preference to short CRAPS descriptions which will generate a high proportion of short successful solution sequences and few long or unsuccessful sequences.

The preliminary investigations we did suggested that the execution time of a completely declarative program will usually be too long to permit a solution except in the simpler cases. Accordingly, the solution sequences are provided by a human expert running the program in "training mode" in which its decisions are observed and corrected when necessary. Further, the human expert is aware of the structure of the program. In subsequent generalization of this approach, we anticipate the necessity of using techniques similar to those of Davis [9], which will enable the trainer to deal only with the external behavior of the program.

We also have deliberately chosen to exclude information about sequences which end in failure. It is clear that, as found by Winston and others [3,21,41,49], counterexamples will be extremely valuable, However, as the reader will note below, even the simpler problem we study poses considerable technical difficulties and it was our feeling that a clearer picture would emerge from the simpler approach.

Furthermore, the experiments described below suggest that useful results can be obtained without counterexamples.

In our initial analysis, the meta-rule construct appeared to be the most ill-defined aspect of our approach. Subsequent experimentation provided us with insight as to how to define them and, consequently, how to automatically construct them. Accordingly, we decided on two directions in which to test our approach. In the first case, we attempt to infer an accurate description of several sequences which would not require the use of meta-rules. We have provided an example of this test case in Section 3.1. Alternatively, we construct an inadequate description, using a single solution sequence, which would force the use of the meta-rules. This allows us to judge the relative power and usefulness of the model we have chosen. Chapter 3 reports the findings of this experiment.

The logical organization of the overall system is outlined in Figure 1.1.

We may describe the technical problem with this approach as inductive inference of a program. The typical inductive inference problem can be formalized as follows (see for example (2)). Given a finite alphabet of symbols, Σ, and two finite subsets S, $T \subseteq \Sigma^*$ (where Σ^* denotes the set of all finite length strings composed of elements from Σ), what is the smallest positive integer k such that there is a program P (in any suitably powerful model of computation) of length equal to k (according to some well defined metric), such that if $L \subseteq \Sigma^*$ is the language accepted (or generated) by P, then $S \subseteq L$ and $T \cap L = \emptyset$? Alternatively, what is the minimum length program P, such that if $L \subseteq \Sigma^*$ is the language accepted (or generated) by P, then $S \subseteq L$ and $T \cap L = \emptyset$? Since we ignore counterexamples (T), our approach to the general problem is to attempt to construct <u>any short program P which generates all of the members of the sample set most of the time</u>. In other words, we attempt to construct a minimal length program which accurately characterizes the given sample. In formal terms, infer the shortest (nondeterministic) program P such that $S \subseteq L$, where L is the language generated by P, and when P is run randomly (making random choices at decision points in the program) for at most some large number of steps then

$$\frac{H}{G2^n}$$

-22-

Figure 1.1 System Organization.
==

```
┌─────────────────────┐                    ┌─────────────────────┐
│ Production System    │                    │ Human Expert:       │
│ for Domain X         │                    │ Guidance            │
└─────────────────────┘                    └─────────────────────┘

        ┌─────────────────────┐          ┌─────────────────────┐
        │     PROSYS:         │          │                     │
        │ PS Interpreter      │─────────▶│ PS Output           │
        └─────────────────────┘          └─────────────────────┘

                ┌─────────────────────┐
                │ Trace Sequences     │
                └─────────────────────┘

                ┌─────────────────────┐
                │     CRAPS           │
                │  compilation        │
                └─────────────────────┘

┌─────────────────────────┐              ┌──────────────────────────────┐
│ CRAPS description:      │              │ Meta-Rules:                  │
│ Heuristics for X        │              │ Aid to CRAPS description      │
└─────────────────────────┘              └──────────────────────────────┘
```
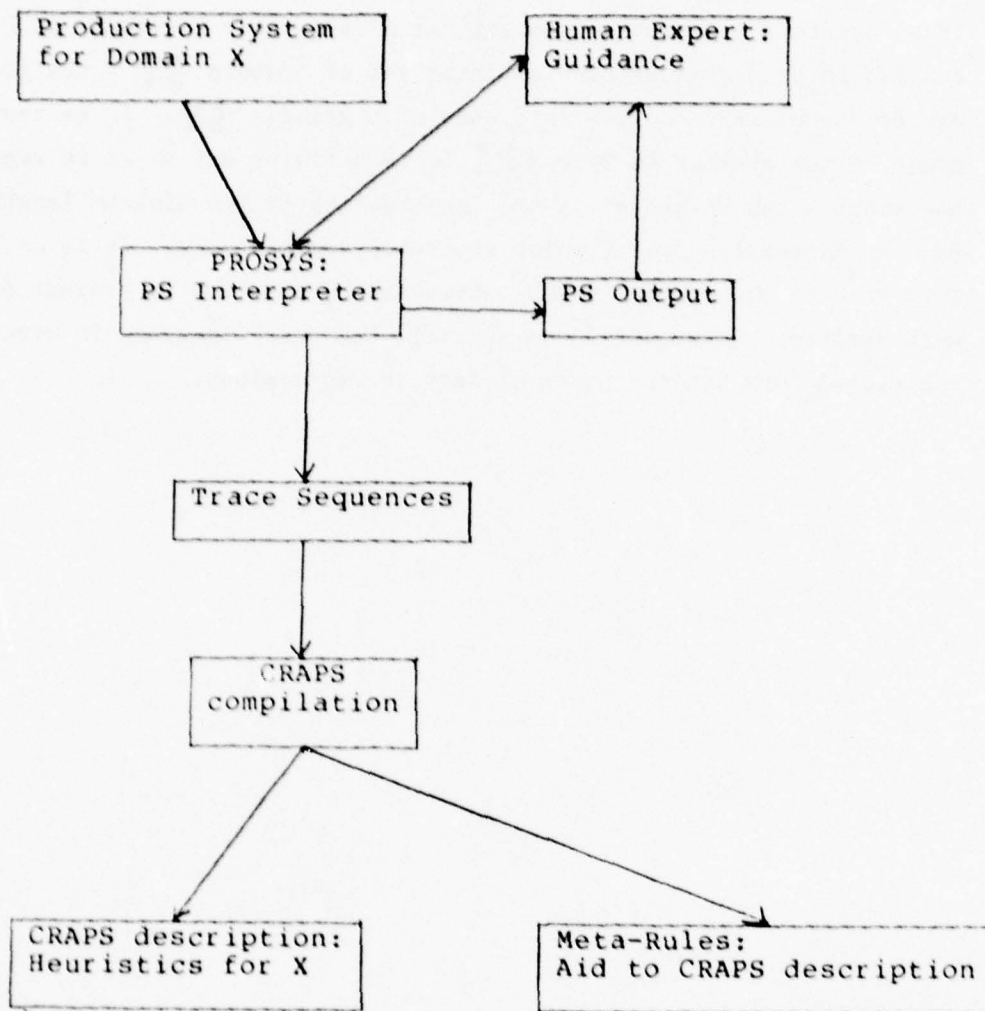
Both are fed back to the PS Interpreter for subsequent
problem solution.

is maximized, where H is the number of strings generated by P that appear in S, G is the number of trials and n is the length of the program (in bits). The formula above captures the notion of the program P generating members of the sample set <u>most of the time</u> with the tradeoff that a program P' whose length is one more than P will be preferred if P' generates strings in S twice as often.

There are several alternative formalizations of the general problem if we consider different computational models. For instance, if P were defined to be a grammar over a finite set of symbols [23] , the problem can be viewed as inductive inference of a grammar [13] . If we restrict the power of the grammar to Type 3 [23] (S is a finite set as so is regular), the problem can be stated as the construction of the minimum length Regular Expression over Σ which generates the language. It is in fact this view of the problem which motivates the solution we present in the next chapter. As we shall see shortly, the CRAPS language in even a restricted form has the power of Regular Expressions.

1.3.1 A Complete Example

To demonstrate our approach, we will discuss a less difficult problem with an algorithmic solution: nondeterministic Binary Tree Traversal [27]. The PS program in Figure 1.2 has the ability to scan a binary tree in any order. No control information is specified, i.e., the productions encode all of the relevant information about scanning binary trees without specifying direction.

A binary tree is represented in WM by the following data format: The root of the tree is represented by (ROOT =X) where =X can be bound to any symbol. If node B has a left son A, it is represented by (LEFT B A) and similarly (RIGHT B C) represents C as the right son of B. The father B of a node A is represented by (FATHER B A). The current node being scanned is (NODE =X). Nodes can be printed only once, and so if node A has been scanned and printed, then (ALREADY-P A) is deposited in WM.

Several binary trees were placed in WM and then the program was initiated and directed by a human to perform an in-order scan of each tree. A trace for each execution was produced (see Figure 1.3). The entire set of traces was analyzed to produce the CRAPS description presented in Figure 1.4.

In this particular case the CRAPS description is a precise definition of an in-order balanced binary tree scan program. That is, given a WM specifying a balanced binary

- 25 -

tree, the shortest execution sequence of the program which
would result in an in-order print of the nodes would belong
to the set of sequences specified by the CRAPS description;
furthermore, the CRAPS description provides enough informa-
tion to determine which production should be fired at every
point in the execution of the program. Thus a nondeterministic
program has in effect been reduced to a deterministic
program.

In general, we cannot assume that the CRAPS description
will be as effective as this. With poor training sequences,
or incorrect analyses of execution traces, CRAPS descrip-
tions may imply heuristics which are not helpful, or which
are helpful for some examples and harmful in others. We
discuss these points more fully in the last chapter.

Figure 1.2 Binary Tree Scanning PS.
======================================

START-POINTING
[(ROOT =X) - (NODE ! =) --> (NODE =X)]

GO-LEFT
[(NODE =X) $ =Cl (LEFT =X =Y) --> (<delete> =Cl) (NODE =Y)
                                                    (FATHER =X =Y)]

GO-RIGHT
[(NODE =X) $ =Cl (RIGHT =X =Y) --> (<delete> =Cl) (NODE =Y)
                                                    (FATHER =X =Y)]

CAN-T-GO-LEFT
[(NODE =X) - (LEFT =X ! =) --> DUMMY)

CAN-T-GO-RIGHT
[(NODE =X) - (RIGHT =X ! =) --> DUMMY)

PRINT
[(NODE =X) - (ALREADY-P =X) --> (NODE =X) (<write> =X)
                                            (ALREADY-P =X)]

GO-UP
[(NODE =X) $ =Cl (FATHER =Y =X) --> (<delete> =Cl) (NODE =Y)]

STOP
[(NODE =X) $ =Cl (ROOT =X) --> (<delete> =Cl) (<halt> FIN)]

Figure 1.3 Execution Traces from Binary Tree PS Run.
==

An exact description of the contents of the trace will be
deferred until chapter 2. A trace is a sequence of lists; the
first item in the list is the rule applied at that point in the
execution, followed by a list of the rules which were also
applicable, but which were not selected for execution.

<<
(START-POINTING NIL (1) (2 2))
(PRINT (STOP CAN-T-GO-RIGHT CAN-T-GO-LEFT) (1 2) (3 4))
(STOP (CAN-T-GO-RIGHT CAN-T-GO-LEFT) (1 4) (5 6))
>>
<<
(START-POINTING NIL (9) (10 10))
(GO-LEFT (STOP PRINT GO-RIGHT) (9 10) (11 13))
(PRINT (GO-UP CAN-T-GO-RIGHT CAN-T-GO-LEFT) (12 13) (14 15))
(GO-UP (CAN-T-GO-RIGHT CAN-T-GO-LEFT) (12 15) (16 17))
(PRINT (STOP GO-LEFT GO-RIGHT) (9 17) (18 19))
(GO-RIGHT (STOP GO-LEFT) (9 19) (20 22))
(PRINT (GO-UP CAN-T-GO-RIGHT CAN-T-GO-LEFT) (21 22) (23 24))
(GO-UP (CAN-T-GO-RIGHT CAN-T-GO-LEFT) (21 24) (25 26))
(STOP (GO-LEFT GO-RIGHT) (9 26) (27 28))
>>
<<
(START-POINTING NIL (14) (15 15))
(GO-LEFT (STOP PRINT GO-RIGHT) (14 15) (16 18))
(GO-LEFT (GO-UP PRINT GO-RIGHT) (17 18) (19 21))
(PRINT (GO-UP CAN-T-GO-RIGHT CAN-T-GO-LEFT) (20 21) (22 23))
(GO-UP (CAN-T-GO-RIGHT CAN-T-GO-LEFT) (20 23) (24 25))
(PRINT (GO-UP GO-LEFT GO-RIGHT) (17 25) (26 27))
(GO-RIGHT (GO-UP GO-LEFT) (17 27) (28 30))
(PRINT (GO-UP CAN-T-GO-RIGHT CAN-T-GO-LEFT) (29 30) (31 32))
(GO-UP (CAN-T-GO-RIGHT CAN-T-GO-LEFT) (29 32) (33 34))
(GO-UP (GO-LEFT GO-RIGHT) (17 34) (35 36))
(PRINT (STOP GO-LEFT GO-RIGHT) (14 36) (37 38))
(GO-RIGHT (STOP GO-LEFT) (14 38) (39 41))
(GO-LEFT (GO-UP PRINT GO-RIGHT) (40 41) (42 44))
(PRINT (GO-UP CAN-T-GO-RIGHT CAN-T-GO-LEFT) (43 44) (45 46))
(GO-UP (CAN-T-GO-RIGHT CAN-T-GO-LEFT) (43 46) (47 48))
(PRINT (GO-UP GO-LEFT GO-RIGHT) (40 48) (49 50))
(GO-RIGHT (GO-UP GO-LEFT) (40 50) (51 53))
(PRINT (GO-UP CAN-T-GO-RIGHT CAN-T-GO-LEFT) (52 53) (54 55))
(GO-UP (CAN-T-GO-RIGHT CAN-T-GO-LEFT) (52 55) (56 57))
(GO-UP (GO-LEFT GO-RIGHT) (40 57) (58 59))
(STOP (GO-LEFT GO-RIGHT) (14 59) (60 61))
>>

Figure 1.4 Abbreviated (and 'pretty-printed') CRAPS output for
Binary Tree PS example.

A formal specification of the CRAPS language is presented
in Chapter 2. A CRAPS description is a sequence of lists
enclosed in (double) pointed brackets (<< >>). Each list con-
tains the name of the rule to apply or a control operation to be
applied to a CRAPS (sub-)sequence. Control operations are en-
closed in square brackets ([ ]).

```
<<Start-pointing
  [If (Go-left & Stop & Print & Go-right)
      Then
      <<[Repeat (While (Go-left & Go-right))
          <<[Repeat (While (Go-left & Go-right))
                    (Until (Can-t-go-left &
                                Can-t-go-right))
              <<Go-left>>]
            [Repeat (While (Can-t-go-left &
                                Can-t-go-right))
                    (Until (Go-left & Go-right))
              <<Print
                [Repeat (While (Go-up))
                        (Until (Print))
                  <<Go-up>>]
                Print
                Go-right>>]>>]>>]
  Print
  [If (Go-up & Can-t-go-right & Can-t-go-left)
      Then
      <<[Repeat (While (Go-up))
                (Until (Stop))
          <<Go-up>>]>>]
  Stop>>
```

1.4 Related Research

Several similar problems have been investigated with
approaches and scope overlapping with ours. We have
attempted below to categorize these projects in the two
areas of knowledge representation and inductive inference.

1.4.1 Knowledge Representation

Recently, many researchers have focused on the repre-
sentation of knowledge in a purely declarative form with a
(nearly) independent procedural component. Cohn [7], uses
a control language very much like CRAPS to specify the
major steps of the deductions in a formal proof system
necessary to prove the correctness of a program. Sickel [39]
and Deliyanni and Kowalski [10], describe an approach
which automatically constructs 'links' between clauses
or formulas in a formal proof system which provides guidance
during the proof process. Both are essentially syntactically
oriented although it can be argued that the latter is more
semantically oriented.

Rychener [36] and Kibler [25] describe approaches in
which the control information is more closely bound to the
declarative representation, in both cases Production Systems.
In effect, meta-rules are compiled into the productions

(in the form of 'control elements' in Rychener's case, and 'directions' in Kibler's case) such that a prescribed set of productions can communicate with each other (through data deposited in WM) and effect a variety of control structures. Kibler's work is the only report known to the author which specifically describes an approach to automatically generating (and compiling) meta-rules.

Davis' **work** [9] (specifically Chapter 7 of his thesis on strategies) laid the ground work for this thesis. The meta-rules he defined for the MYCIN system are an independent collection of rules to be applied when the next step of the deduction is not clear. The meta-rules we define in this thesis are of exactly this nature. However, his meta-rules are limited in scope (i.e. problem or data specific) and explicitly acquired (and fine-tuned) by a human expert. We present an approach to automatically generating a set of meta-rules based on the transformations of the conflict set during execution without human intervention (although the system is trained by a human from examples). To date, no research has been reported which combines automatic program generation with the production system, meta-rule methodology.

Quite similar to our overall goal is the report by Fikes, Hart and Nilsson [14]. They introduced the notion of 'robot-plans' in relation to the STRIPS problem-solving system, a formal proof system. Sequences of operator appli-

cations used in the solution of sample problems were stored
for possible guidance (or planning) in solving subsequent
problems. Data items that were used by and propagated
through sequences of operators were stored in tables, and
later used as triggers for applying the known sequences.
The work we report stresses the importance of a deeper
analysis of the sequence of operator applications.

1.4.2 Inductive Inference

It is very difficult to characterize much of the work in inductive inference, partly because of the scope of this rather large field. We have grouped several fields usually associated with artificial intelligence research under this topic: automatic programming, program construction, inductive inference and learning. As is often the case with such categorization, the boundary lines between these fields are often blurred or nonexistent.

The approaches to automatic program construction can be loosely termed transformational approaches. That is, starting with a high level or abstract description of what the target program should do on certain inputs, transform the specification into an executable program satisfying the specification. The specifications are in a variety of forms, including predicate calculus, set theory and lists of input/output pairs. It appears that the more difficult problems are those that have purely syntactic input (i.e. uninterpreted symbols in the specification) and only the relationship between input and output (i.e. no intermediate states). Many such systems have been developed, all with limited success. See for example Lee, Gerhart, and De Roever [28], Summers [42] and Waldinger and Lee [46]. Summers' work is particularly interesting since it is well defined and couched in a very rigorous mathematical theory of LISP programs.

The solutions to problems that have been studied which allow interpretation and intermediate states can be characterized by ad hoc, or of the heuristic search (hypothesize and test) variety. For example, Abrahams [1] reports on a system that predicts sequences of interpreted atoms based on a collection of ad hoc heuristics. Phillips' [33] work, as part of the larger PSI automatic programming project at Stanford, is based on a collection of ad hoc heuristics for constructing LISP programs.

The work we report on allows for intermediate states (the entire solution sequence) and uninterpreted symbols (only equality relates components). We provide algorithms for the analysis of sequences which we view as good approximations employing helpful heuristic rules, and are not of the more ad hoc variety. The algorithms we use do not search a large space of programs looking for a minimum solution, but instead produce one output directly (although searching of the components of the sequences is of course necessary in the analysis). We view the meta-rule construct we use as a fine-tuning mechanism, correcting the generated program whenever necessary. The optimal solution may not be required if there is an appropriate set of 'patch-ups' that allow for a more accurate solution.

Many projects have been reported which use the 'hypothesize and test' paradigm of program generation (or grammatical inference). Biermann [5] is one such system

which offers a simple but elegant heuristic search algorithm
which generates Turing machines from uninterpreted inter-
mediate states. As he points out, the model of learning
which he describes has been studied by many workers in
grammatical inference, for example, Feldman [13],
Hedrick [21], Solomonoff [40], and Waterman [47]
(to a certain extent). Hewitt's work with PLANNER [22]
and the work of Schmidt and others on the Plan Recognition
Problem [38] are also of this form. It seems apparent
that such approaches may be effective only in the presence
of very sophisticated heuristics.

Angluin's work [2,3,4] requires special mention.
It is not very often that problems in artificial intelligence
are defined with a high degree of rigor. She has stated
the problem of Minimum Inference of Regular Expressions
and proved its NP-completeness (see Garey and Johnson [17]).
Given a finite alphabet of symbols Σ, two finite subsets
$S, T \subseteq \Sigma^*$, and a positive integer K, is there a regular
expression E over Σ that has K or fewer occurrences of
symbols from Σ and such that, if $L \subseteq \Sigma^*$ is the language
represented by E, then $S \subseteq L$ and $T \subseteq \Sigma^* - L$? In fact
she also showed that this problem remains NP-complete if '$\cup$'
operations or Kleene star (*) operations are not allowed
in the regular expressions. (This corresponds to disallow-
ing alternation and repetition, respectively, in our analysis.)
Angluin [3] elected to solve a simpler problem, namely

finding a pattern common to a set of strings where the
patterns defined are different from regular expressions.
Since it is believed by many researchers that NP-completeness
implies intractability (see Garey and Johnson [17]), we
have elected to approximate the solution of the more general
problem. Notice that T as defined above corresponds to
counterexamples which we do not use. As mentioned earlier,
Winston's [49] work stresses the importance of counter-
examples. We mention Hunt et al. [24] simply for its
similarity to Winston's work.

In a more general setting, many researchers have
studied inductive inference within formal syntactic systems.
These approaches are based on formal inductive rules of
inference. Other related work includes Meltzer [31],
Plotkin [34], Popplestone [35] and Sperling [41].

Finally, in fields quite different from those mentioned
here, there is some work strikingly similar to the problems
we consider in this thesis. Fosdick and Osterweil [16]
describe an approach to program testing based on a data flow
analysis which constructs 'patterns of variable usage',
which are essentially regular expressions. A classification
scheme of such patterns suggests which paths in the
program contain anomalies, and can then be used to predict
errors in the program. Habermann [18] in quite a different
application describes the use of 'path expressions', which is
essentially an independent (yet integrated) collection of informa-
tion used to control the synchronization of concurrent processes.

CHAPTER 2.   THE CRAPS LANGUAGE

In this chapter we formally outline the specification of the CRAPS language and then present the algorithms which compile CRAPS descriptions. The UT LISP code which implements the algorithms has not been included in this thesis.

2.1   A Formal Specification

As indicated above, the CRAPS language provides a semantic framework with which to specify or describe sequences of rule applications in the execution of the nondeterministic program. The basic primitive of CRAPS is called a *unit*. A unit specifies either a rule application with preconditions, in which case it is called a *simple* unit, or a control operation applied to a sequence of units. The control operations, which are formally defined in the next section, are (implicit) Concatenation of units producing sequences, Repetition of a sequence controlled by simple Boolean assertions in Disjunctive Normal Form (DNF), Alternative or conditional selection of a sequence from a set of sequences and the Permutation of a set of sequences. (From which we have derived the acronym CRAPS.)

The CRAPS operators correspond to various control primitives of conventional programming languages. Concatena-

tion corresponds to sequential execution of statements
and the repetition operator corresponds to iteration
statements with continuation and termination conditions.
The alternation operator specifies alternative sequences
of actions much like an ALGOL-68 'case' statement or LISP
conditional expression. The permutation primitive
represents a form of concurrent execution similar to the
ALGOL-68 'collateral expression'.

2.1.1 A BNF Description of CRAPS

In what follows, parentheses are terminal symbols. Alternatives in the right-hand side of a production are separated by lines and preceded with ::= . The nonterminal <atom> is any (LISP) identifier.

```
<sequence>       ::=<< <unit+> >>
<sequence+>      ::=   <sequence> <sequence+>
                 ::=   <sequence>
<unit+>          ::=   <unit> <unit+>
                 ::=   <unit>
<unit>           ::=   <simple-u>
                 ::=   <repetition-u>
                 ::=   <permutation-u>
                 ::=   <atom-u>
                 ::=   <alternation-u>
<simple-u>       ::=    (<atom> <dnf>)
<atom*>          ::=   <atom> <atom*>
                 ::=
<repetition-u>::=    (R.* <sequence>
                     (W.* <dnf>)
                     (U.* <dnf>) )
<dnf>            ::=   ( OR.* ( <atom*>) <disjuncts+>)
                 ::=   ( <atom*> )
<disjuncts+>     ::=   ( <atom*> ) <disjuncts+>
                 ::=   ( <atom*> )
<permutation-u>  ::=(P.* <sequence ><sequence+ >)
<atom-u>         ::=   <atom>
<alternation-u>::=    (A.* <cond> <cond*>)
<cond>           ::=   (<sequence> <dnf>)
<cond*>          ::=   <cond> <cond*>
                 ::=
```

The syntax specified is used in the actual implementa-
tion. However, a 'pretty-printed' version of the language
is used in the machine output contained in this thesis.
Consequently, R.* is replaced by REPEAT, P.* by PERMUTE,
and A.* by an IF THEN ELSE form, each enclosed in square
brackets (see Appendix 2 and Figure 1.4).

2.1.2 Definition of the CRAPS Operators

Cambridge notation is used for the operators except
for concatenation which is implicitly specified as a
sequence of units.

1. Concatenation of units.

Let $s_1, s_2, \ldots, s_n$ be units. Then the sequence
$S = <<s_1 \ s_2 \ \ldots \ s_n>>$ is the concatenation of   the units s_i,
$i = 1, \ldots, n$. (Throughout the remainder of this paper,
sequences will be represented by a sequence of objects
enclosed in pointed brackets, $<<$ and $>>$, and separated
by blanks.

2. Concatenation of sequences.

Let $S_1 = <<s_{11} \ s_{12} \ \ldots \ s_{1m_1}>>$

$\qquad S_2 = <<s_{21} \ s_{22} \ \ldots \ s_{2m_2}>>$

$\qquad \vdots$

$\qquad S_j = <<s_{j1} \ s_{j2} \ \ldots \ s_{jm_j}>>$

Then $\qquad <<S_1 \ S_2 \ \ldots \ S_j>> =$

$<<s_{11} \ s_{12} \ \ldots \ s_{1m_1} \ s_{21} \ s_{22} \ \ldots \ s_{2m_2} \ \ldots \ s_{j1} \ s_{j2} \ \ldots \ s_{jm_j}>>$

In the obvious way, we denote

$$S^n = \begin{cases} <<S^{n-1} \ S>> & \text{if } n \geq 1 \\ \text{the null sequence} & \text{if } n = 0 \end{cases}$$

3. Repetition of a sequence.

Let S_1 be as in 2 above. Then

$$(R.* \ S_1 \ (W.* \ c_1) \ (U.* \ c_2)) = S_1^n$$

where $n \geq 1$ is chosen so that repetition terminates as soon as c_1 is false or c_2 is true. The conditions c_1 and c_2 are simple Boolean assertions of rule applicability written in DNF (see 6 below).

4. Alternation of a set of sequences.

Let $S_1, S_2, \ldots, S_j$ be as in 2 above.
Let $U = (A.* \ (S_1 \ c_1) \ (S_2 \ c_2) \ \ldots \ (S_j \ c_j))$, $j \geq 1$,
and c_i , $i = 1, \ldots, j$ are simple Boolean assertions in DNF. Then

$$U = \begin{cases} \text{if } c_1 \text{ is true then } S_1 \\ \quad \text{else if } c_2 \text{ is true then } S_2 \\ \quad\quad \text{else } \ldots \\ \quad\quad\quad \text{else if } c_j \text{ is true then } S_j \\ \quad\quad\quad\quad \text{else } S^0. \end{cases}$$

5. Permutation of two or more sequences.

Let $S_1, S_2, \ldots, S_j$ be as in 2 above. Then

$$(P.* \ S_1 \ S_2 \ \ldots \ S_j) =$$
$$\left\{ <<s_{i_1 k_1} \ s_{i_2 k_2} \ \cdots \ s_{i_\ell k_\ell}>> \ \middle| \ \ell = \sum_{t=1}^{j} m_t \ , \text{ and} \right.$$
$$\left. \text{if } i_r = i_q \text{ then } k_r < k_q \right\}.$$

The permutation operator is more like a <u>shuffle operator</u>, that is the argument sequences can be merged in any order as long as the ordering within each sequence is maintained.

- 42 -

6. DNF conditions.

Let $C = $ (OR.* $(a_{11} \ a_{12} \ \cdots \ a_{1n_1})$

$$\vdots$$

$(a_{k1} \ a_{k2} \ \cdots \ a_{kn_k})$)

Then C is true if $1 \leq \exists i \leq k$ such that $1 \leq \forall j \leq n_i$, a_{ij} is active, otherwise C is false. If $C = (a_1 \ a_2 \ \cdots \ a_k)$, then C is true if $1 \leq \forall j \leq k$, a_j is active.

A simple unit is the most basic primitive of CRAPS. It is composed of two parts, the name of a rule to apply and a DNF expression that must be sa' ᶜied (i.e. evaluate to true) in order to execute the associated rule name.

The (programmed) interpretation of the above definitions is as follows: A sequence of units specifies sequential application of each of the component units. The repetition of a sequence is much like a **PL/I** **do** loop; apply the argument sequence one or more times while the condition c_1 is true and c_2 is false.

The alternation operator selects the first sequence to apply whose corresponding condition, c_i , is true. The permutation of a set of sequences specifies that each of its argument sequences are to be applied in any order or merged in any order; sequential execution of the appended arguments is chosen.

Finally, in the execution of a production system, whenever a DNF condition is to be evaluated, the condition

- 43 -

is true if any of its conjuncts are true; otherwise it is
false. A conjunct is true if each of the   productions
listed are matched by data from working memory at the time
the condition is being evaluated. Examples of these
definitions follow.

2.1.3 Examples

(1) When executing the simple unit

(GO-LEFT (OR.* (STOP PRINT) (GO-RIGHT))),

if the productions STOP, GO-RIGHT and GO-LEFT are active,

then the rule named GO-LEFT would be executed next.

(2) (R.* << (GO-RIGHT (PRINT))

(PRINT (OR.* (GO-RIGHT) (STOP)))>>

(W.* (GO-RIGHT))

(U.* (STOP)))

produces the sequencing:

<< (GO-RIGHT (PRINT))

(PRINT (OR.* (GO-RIGHT) (STOP)))

(GO-RIGHT (PRINT))

(PRINT (OR.* (GO-RIGHT) (STOP)))

.
.
.   >>

continually while the production GO-RIGHT is active
and STOP is inactive. The conditions are tested when
the end of the argument sequence is reached. Notice,

however, that the preconditions of the units contained
in the argument sequence must still be satisfied
when executing the argument sequence. The details
are in Section 2.1.4.

(3)   (P.* <<(START-ENGINE ( )) (SHIFT-FORWARD ( ))>>
          <<(TURN-ON-RADIO ( )) >>)

could produce 3 sequences:

{<<(START-ENGINE ()) (SHIFT-FORWARD ()) (TURN-ON-RADIO ())>>
  <<(START-ENGINE ()) (TURN-ON-RADIO ()) (SHIFT-FORWARD ())>>
  <<(TURN-ON-RADIO ()) (START-ENGINE ()) (SHIFT-FORWARD ())>>

                                                          }

In a parallel environment (two arms for a robot) both
sequences could be done simultaneously. In the actual
implementation of the language, the argument sequences
are concatenated and executed in order.

(4)   (A.* (<<(**EAT** ( ))>> (THERE-IS-FOOD))
          (<<(THANK-HOST ( ))>> (PLATE-EMPTY)))
produces the sequencing <<(**EAT** ( ))>> if the production
THERE-IS-FOOD is active, otherwise, <<(THANK-HOST ( ))>>
if the production PLATE-EMPTY is active, or it produces
no sequencing at all.

2.1.4 Implementation

The implementation of the operators is very straight-
forward. The PS interpreter maintains a stack of units
corresponding to the CRAPS description that is in control
of the sequencing. Successive units are popped from the
stack and applied. If the stack is empty, or the
precondition of a simple unit is false, the meta-rules
are called. The algorithm of figure 2.1 is invoked
whenever a rule is to be selected from the conflict set.
The details of the meta-rule implementation can be found
in the next section.

Figure 2.1 Implementation of CRAPS.
======================================

```
repeat forever; Apply the rule selected as follows:

    case;

        (Stack Empty):
          Set P to meta-rule choice;
          If P not empty then select P from conflict set;
          else stop;   end if;

        (Simple unit on stack):
          If the DNF of the unit is true and the rule name is active
             then
               pop the stack;
               select the rule name from the conflict set;
             else
               set P to the meta-rule choice;
               if P is not empty then select P from conflict set;
                  else pop the stack; end if;
             end if;

        (Permutation unit on stack):
          pop the stack;
          push the appended argument sequences on stack;

        (Repetition unit on stack):
          if the While component, and the Until component are both
             empty then
             pop the stack;
             push argument sequence on stack; /* executed once */
           else
             if While condition is true and Until is false then
               push argument sequence on stack;
              else
               pop the stack;
             end if;
           end if;

        (Alternation unit on stack):
          Scan each argument of unit;
            if the DNF condition is true then
              pop the stack;
              push the corresponding argument sequence;
            end if;
          end scan;
          if none were true then pop the stack;
    esac;

  end repeat;
```

2.2 Meta-Rules

There are four types of meta-rules which assist a CRAPS description in controlling a PS. It is the simple unit which actually selects the next rule to fire on each cycle (the higher level control units produce sequences of simple units), and if in the event that the DNF expression evaluates to false or the specified rule is not active, the meta-rules are called upon to suggest a list of rule names to try. For example, suppose that the simple unit (A (B C)) is in control, E was the previously fired production, and the current conflict set of rules is {B D}. This situation may be described as:

(1)   A and C **should be active**

(2)   D should (perhaps) be inactive

(3)   {B D} is currently active

(4)   E was just fired.

Accordingly, the meta-rules which have been implemented are designed to deal with the four cases listed. In each case, a meta-rule may suggest a list of rules to try in that situation. The suggestions are weighted since a rule may be suggested several times by different meta-rules. Each rule is scanned and its LHS tested; if it evaluates to true, the corresponding RHS is executed. The exact definitions of the meta-rules appear in the next section.

2.2.1 A BNF Description of the Meta-Rules

The same conventions are followed as in 2.1.1.

```
<meta-rule>    ::=   [<m-LHS> --> <m-RHS>]

<m-LHS>        ::=   (Want-active (<atom*>))

               ::=   (Want-inactive (<atom*>))

               ::=   (Currently-active (<atom*>))

               ::=   (Just-fired (<atom*>))

<m-RHS>        ::=   (Try-to-fire (<atom*>))

<atom*>        ::=   <atom> <atom*>

               ::=   <atom>
```

The following is an example of a meta-rule:

```
        [(Want-active(A  C)) -->

                     (Try-to-fire(G))]
```

Others can be found in Figure 3.3.

2.2.2 Definition of the Primitives

Let P' be a list of rule names

J   be the rule fired on the previous cycle

D   be the union of all the names appearing in the DNF

of the current simple unit in control

U   be the rule specified by the current simple unit

C   be the current set of active rules.

The primitive function Want-active is a Boolean function
which tests the current state of the CRAPS description and
the currently active set of rules. It is defined as follows:

- 49 -

$$(\text{Want-active } P') = \begin{cases} \text{true if } P' \subseteq D \cup \{U\} \\ \text{else false} \end{cases}$$

Similarly, the other primitive functions are defined as follows:

$$(\text{Want-inactive } P') = \begin{cases} \text{true if } P' \subseteq C - (D \cup \{U\}) \\ \text{else false} \end{cases}$$

$$(\text{Currently-active } P') = \begin{cases} \text{true if } P' \subseteq C \\ \text{else false} \end{cases}$$

$$(\text{Just-fired } P') = \begin{cases} \text{true if } J \subseteq P' \\ \text{else false} \end{cases}$$

The definitions are very simple and so the details of the implementation are not included.

The function Try-to-fire deposits a subset of its argument list of rule names in a master list (TRYLIST) maintained by the interpreter. All suggested rules must be active. The most often suggested rule is selected for execution; in the case of ties, preference is given to the rule name in the current simple unit, U.

Try-to-fire is defined as:

```
Try-to-fire(P'): procedure;

  TRYLIST: =

    TRYLIST ∪ (P' ∩ C); /* but duplicates remain */
  end;
```

2.3 Analyzing Sequences

When running a production system to a successful conclusion, a trace of the actions performed by the PS is generated. A series of these traces, called input trace sequences, are presented to a system which analyzes and produces CRAPS descriptions from them. The input sequences, which are described in the next section, are transformed to a variety of intermediate states until a final CRAPS sequence is constructed.

2.3.1 Input Trace Sequences

The exact form of an input sequence is $<<P_1\ P_2\ \ldots\ >>$ where P_i, called a *trace unit*, is of the form:

$$(P_i'\ (P_{i1}\ P_{i2}\ \ldots\ P_{it_i})\ (N_{i1}\ N_{i2}\ \ldots\ N_{ij_i})\ (M_{i1}\ M_{i2}))\ .$$

P_i' is the name of the rule which was applied on the ith cycle of the execution of the PS during the training session. $(P_{i1}\ P_{i2}\ \ldots\ P_{it_i})$ is the set of rules which matched the data environment during the ith cycle. This set encapsulates the data environment at that point in the execution, and is used to calculate the DNF conditions for the units and the meta-rules.

The third component of the trace unit, $(N_{i1}\ N_{i2}\ \ldots\ N_{ij_i})$, is the set of unique integers associated with the instances of data elements which matched the pattern of the rule P_i'. Finally, $(M_{i1}\ M_{i2})$ is the range of unique integers associated

with all of the data elements produced by the action portion
of the rule P_i'. The last two pieces of information allow for
the construction of a data flow graph.

The trace sequence is an exact history of the execution
of the nondeterministic program and is therefore already
partially ordered by the 'back-dominance relation.' Every
trace unit within the sequence is preceded by those trace
units which produced data for it. This partial ordering
simplifies much of the subsequent analysis of the trace
sequence.

Figure 1.2 is an example of a trace sequence
produced by a run of the binary tree scanning PS.

2.3.2 Notation and Terminology

The algorithms which follow are presented in a SETL-
like language which makes use of n-tuple and set notation.
Lower case symbols in the code represent key words with
obvious meanings. All sequences will be represented by
vectors whose enclosing brackets are < and > . Each item
of a sequence is referenced by integer indices. The null
sequence is represented by the symbol omega.   Calls to
subroutines will be denoted by prefix notation with arguments
enclosed in parentheses. The binary operators used (for
example +) have various interpretations depending upon
the type of data structure supplied to them. For example,
+ can represent set union, vector concatenation or integer
addition.

The input trace units are transformed to a variety of
intermediate forms until a final sequence of CRAPS units
is produced. A trace unit will be called a T-unit; a
sequence of trace units, a T-SEQUENCE; a sequence of units
in intermediate form an I-SEQUENCE; and a CRAPS sequence
a C-SEQUENCE. The intermediate, simple, atom, permutation,
repetition and alternation units will be termed,
respectively, I-, S-, A-, P-, R-, and ALT-units. Each type
of unit will be represented by a record type data structure
with named components.

Declarations:

```
T-unit: ( PNAME: atom,
          TRUE-PS: set of atoms,
          IN-LIST: set of integer,
          OUT-LIST: [1:2] integer,
          SONS: [1:] integer);

I-unit: ( NUMBER-PNAME:(NODENUM: integer,
                        PNAME: atom),
          TRUE-PS: as in T-unit,
          IN-LIST: as in T-unit,
          OUT-LIST: as in T-unit,
          SONS: as in T-unit);

S-unit: ( NUMBER-PNAME: as in I-unit,
          TRUE-PS: as in T-unit);

A-unit: ( NAME: atom,
          ORDERING: integer,
          EQUIVALENTTO: A-unit,
          SUBSEQNC: P-unit);
```

Each P-, R- and ALT-unit begins with the signifier

P.*, R.* and A.*, respectively. The remaining components

of each are:

```
P-unit: ( SUBSEQUENCES: [1:(>=2)] C-SEQUENCE);

R-unit: ( LENTH: integer,
          C-SEQ: C-SEQUENCE,

          WHILE: (atom= 'W.*',
                  DNF: Boolean expression),
          UNTIL: (atom = 'U.*',
                  DNF: Boolean expression) );

ALT-unit: ( [1:(>=2)] COND: ( C-SEQ: C-SEQUENCE,
                              DNF: Boolean expression));
```

In the discussion of graphs, ordinary graph notation

will be used, Nodes, edges and paths will be referenced

by subscripted v's, e's, and p's, respectively.

2.3.3 Algorithms

2.3.3.1 Top level control

The top level algorithm repeatedly reads in T-SEQUENCES and processes each one by calling a succession of routines. It begins by constructing a graph which identifies all of the data dependencies within a sequence. This graph is

then linearized by the permutation routine to obtain a preliminary description of the sequence. The repetition algorithm is then called to produce descriptions which possibly contain R-units. All sequences read in are processed in this fashion, producing a set of descriptions which are input to the alternation detection algorithm. The final output is a CRAPS description. See Figure 2.2 for complete details of this algorithm. An example of an actual run of the system is exhibited in Appendix 2.

Note that constructing units in this order prohibits the appearance of alternation within a repetition and permutation. Because of the difficulty of the pattern analysis considered here, less powerful CRAPS descriptions are generated than can be specified in the language. A discussion of these points is deferred until Chapter 3.

```
Figure 2.2: Top level control.
================================

DESCRIBE(): procedure;

NODE-COUNT: integer; /*assignment of unique node numbers*/
PCOUNT: integer; /*assignment of artificial ordering*/
PROD-NAMES: set of atoms; /*production names*/
PSTARNAMES: set of A-units;
VISITED: [1:] I-SEQUENCE; /*indexed by node numbers and used
                      for linear sequence constructed from DAG*/
NODEPTRS: [1:] I-unit; /*indexed by node numbers*/
ND: [1:] integer; /* numbers assigned to nodes of
                      DAG for Tarjan's algorithm*/
DFSN: [1:] integer; /*depth first spanning tree numbers for
                      the algorithm*/
DFSNUMBER: integer; /*used for values of DFSN*/
SEQ: T-SEQUENCE or I-SEQUENCE or C-SEQUENCE or atom;
DESCS: set of I-SEQUENCE;
ENVIRONS: set of pairs of sets; /*for meta-rule construction*/
MADE-ACTIVE: like ENVIRONS;
MADE-INACTIVE: like ENVIRONS;
JUST-FIREDS: like ENVIRONS;

/* Initialization */

      PCOUNT := 0;
      DFSNUMBER := 0;
      NODE-COUNT := 0;
      PSTARNAMES := {};
      PROD-NAMES := {};
      DESCS := {};
      ENVIRONS := {};
      MADE-ACTIVE := {};
      MADE-INACTIVE := {};
      JUST-FIREDS := {};

/* MAIN INPUT LOOP */

      repeat while( SEQ \= omega ) doing( read(SEQ) );

/* We assume production names have an ORDERING component. */

      if SEQ is a production name then
          SEQ.ORDERING := PCOUNT;
          PCOUNT := PCOUNT + 1;

      else

/* Preparation for processing an input trace sequence. */

          LENTH := length(SEQ);
          VISITED[1:LENTH] := omega;
          NODEPTRS[1:LENTH] := omega;
          DFSN[1:LENTH] := 0;
          ND[1:LENTH] := 0;

/* Assign unique node numbers to each T-unit */

          SEQ := CHANGEP-FIRED(SEQ);

/* Construct the DAG and Meta-rules. */
```

```
                    SEQ := INTERVAL-OF-EXIT(SEQ);

/* Construct permutation units */

                    SEQ := CONSTRUCT-P*(SEQ);

/* Repetition detection. Notice () is sent as the
 second argument since SEQ is not followed by any
 unit.   See REPEATS for a description of this.*/

                    SEQ := REPEATS(SEQ,());
                    NODECOUNT := LENTH + NODECOUNT;
                    DESCS := DESCS +  {SEQ};

               end if SEQ;

          end repeat;

/* FINAL PROCESS */

/* We now collapse all of the sequences into one CRAPS
   description. */

     SEQ := ALTERNATIVES(DESCS);

/* All of the atom units are replaced by their corresponding
  permutations.  The following routine is a simple scanning
  procedure which does the replacement. It is not shown. */

     SEQ := REPLACE-ATOMS(SEQ);

/* We can now remove the numbering of simple units and the
   lengths within the repetitions.  The numbering was
   necessary for duplicates within the permutations. All such
   duplicates can now be removed also.  The following is a
   simple scanning routine which does the removal and is not
   shown.  */

     SEQ := REMOVENUMSANDDUPS(SEQ);

/* The description and meta-rules have been constructed so... */

     print(SEQ,ENVIRONS,MADE-ACTIVE,MADE-INACTIVE,JUST-FIREDS);

end DESCRIBE;
```

2.3.3.2 DAG Routines

The input sequence is scanned and a unique number is assigned to each instance of a rule application in the T-units. A Directed Acyclic Graph (DAG) is then constructed with nodes (indexed by the unique numbers) corresponding to rule applications and edges representing the flow of data from one rule to another. This DAG is constructed in an obvious manner from the IN-LIST and OUT-LIST components of the T-units. It is assumed that the last unit in the sequence signalled success of the problem solution and therefore its corresponding node is interpreted as the unique sink node of the DAG (see page 63 for examples).

The DAG completely specifies all of the data dependencies between rules and therefore all of the possible execution sequences. In general, an arc from node v_i to node v_j indicates a concatenation of the subsequence associated with v_i (leading to and including v_i) with the unit associated with node v_j.

The DAG is first cleansed of irrelevant arcs, called forward arcs. An arc e, from node v_i to node v_j is a forward arc if there is some directed path p from v_i to v_j which does not include e as an intermediate arc. A forward arc specifies that node v_i must precede v_j and that node v_i precedes the nodes on path p. Since the nodes on path p enter v_j, they too must precede v_j. Clearly, the forward arc is redundant and can be removed from the DAG, which prohibits

- 58 -

the construction of irrelevant permutations.

The forward arc removal algorithm is due to R. Tarjan [43,44,45]. It proceeds as follows:

Let DFSN(i) be a Depth-first Spanning Tree numbering assigned to node i beginning the numbering from the sink node of the graph. Let ND(i) be the sum of the numbers assigned to the immediate predecessors of node i. Then there is a path p from v_i to v_j iff DFSN(i) $\leq$ DFSN(j) $\leq$ DFSN(i)+ND(i). Given a node v and the set of entering edges, it is easy to check from this inequality which edges are forward arcs. For any two immediate predecessors, v_i and v_j , if the above inequality holds, then the edge from v_j to v can be deleted. Figure 2.3 contains complete details of the algorithm.

Figure 2.3: DAG construction and cleansing.
===

```
CHANGEP-FIRED(SEQ): procedure;

SEQ: T-SEQUENCE;

    CNTR := 0;

    1 <= forall i <= #SEQ;

/* The PNAME component is replaced by a NUMBER-PNAME
   component producing I-units. */

        SEQ[i].PNAME := <CNTR, SEQ[i].PNAME>;
        CNTR := CNTR + 1;

    end forall i;

end CHANGEP-FIRED;


INTERVAL-OF-EXIT(SEQ): procedure;

SEQ: I-SEQUENCE;

/* The last unit of SEQ is assumed to be the unique sink
   of the DAG to be constructed by this routine.  SEQ is
   partially ordered by the back-dominance relation. */

    #SEQ >= forall i >= 1;
        NODEPTRS[SEQ[i].NODENUM] := SEQ[i];

        #SEQ >= forall j > i;
            if exists n in SEQ[i].IN-LIST such that
                SEQ[j].OUT-LIST[1]
                                <= n <= SEQ[j].OUT-LIST[2]
                then
                SEQ[i].SONS := SEQ[i].SONS
                                    + <SEQ[j].NODENUM>;
            end if;

        end forall j;

/* Meta-rule construction is done at this point, see figure
   2.7.  The definitions in section 2.2 suggest simple set
   expressions between the data environments(conflict sets)
   between successive input units. */

        CONSTRUCT-M-RULE(SEQ[i-1],SEQ[i]);

    end forall i;

    return CLEAN(SEQ);

end INTERVAL-OF-EXIT;


CLEAN(SEQ): procedure;

SEQ: I-SEQUENCE;

/* IN-LIST and OUT-LIST components are no longer useful. */
```

```
                    1 <= forall i <= #SEQ;
                        SEQ[i].IN-LIST := omega;
                        SEQ[i].OUT-LIST := omega;
                    end forall i;

        /* We are now ready to process the DAG after we have reversed
           SEQ so that the sink node is first. */

                    SEQ := reverse(SEQ);


        /* Now we use Tarjan's algorithm to remove forward arcs.*/

                    REMOVEFWDARCS(SEQ);
                    SEQ := reverse(SEQ);
                    return SEQ;

        end CLEAN;


        REMOVEFWDARCS(SEQ): procedure;

        SEQ: I-SEQUENCE;

        /* Tarjan's algorithm. */

                    DFSNUMBER := 1;
                    DEPTH-FIRST-SEARCH(SEQ[1]);

                    1 <= forall i <= #SEQ;

                        if #SEQ[i].SONS >= 2 then

                            forall child1 in SEQ[i].SONS;

                                forall child2 in SEQ[i].SONS - {child1};

                                    if TARJAN-INEQUALITY(child1,child2) then

                                        /* remove the arc , child1 = omega */
                                        SEQ[i].SONS := SEQ[i].SONS - {child1};

                                    end if;

                                end forall child2;

                            end forall child1;

                        end if;

                    end forall i;

        end REMOVEFWDARCS;

        TARJAN-INEQUALITY(N1,N2): procedure;

        N1,N2: integer; /* node numbers */

                    if DFSN[N1] <= DFSN[N2] <= DFSN[N1] + ND[N1]
                     then
                        return true
                    else return false;
                    end if;

        end TARJAN-INEQUALITY;
```

```
DEPTH-FIRST-SEARCH(UNIT): procedure;

UNIT: I-UNIT;

     if DFSN[UNIT.NODENUM] = 0
      then
          DFSN[UNIT.NODENUM] := DFSNUMBER;
          DFSNUMBER := DFSNUMBER + 1;
          V, ND[UNIT.NODENUM] := 1 +
                          +:{DEPTH-FIRST-SEARCH(NODEPTRS[s]) !
                                      forall s in UNIT.SONS};
          return v;
      else return 0
      end if;

end DEPTH-FIRST-SEARCH;
```

2.3.3.3 Permutation Routines

The construction of permutations is based on the observation that if a node has two or more entering edges then the subsequences associated with its predecessors can be permuted. The partial ordering makes this construction straightforward. Using a simple sequential scan, it is assured that the sequences associated with its predecessors have already been constructed when a node is to be processed.

Forward arc removal prevents an irrelevant permutation from being constructed. Consider the following DAG (we have removed the TRUE-PS field for simplicity):
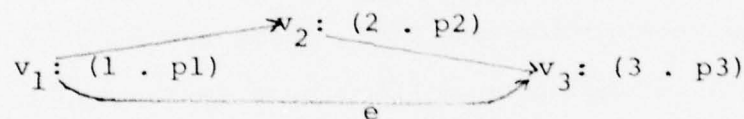
When node v_3 is to be processed, the following subsequence would be constructed: $<< (P.* <<v_1\ v_2>> <<v_1>>)\ v_3>>$. The permutation is obviously irrelevant and if arc e were ignored then the subsequence constructed would be $<<v_1\ v_2\ v_3>>$.

The permutations are further processed by first factoring out common leading subsequences and then applying the repetition detection procedure so that equivalences with other permutations could be more easily recognized.

Consider the following graph:
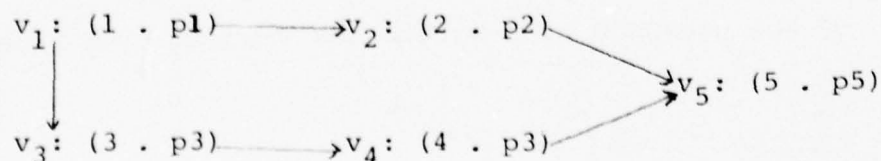
The subsequences associated with nodes v_1 , v_2 , v_3 and v_4
are, respectively, $<<v_1>>$, $<<v_1\ v_2>>$, $<<v_1\ v_3>>$, $<<v_1\ v_3\ v_4>>$.
When processing v_5 the following would be constructed:
$<<(P.*\ <<v_1\ v_2>>\ <<v_1\ v_3\ v_4>>)\ v_5>>$. Each of the subsequences
begins with node v_1. By definition of permutation, this
is equivalent to $<<v_1\ (P.*\ <<v_2>>\ <<v_3\ v_4>>)\ v_5>>$. This
final form is less complex and more closely represents
the control information in the DAG. Notice that if v_3
were instead (6 . P2) we could not factor out nodes v_2
and v_3 since they are distinct instances of applying rule p2
and could not be identified as the same. The final permu-
tation can be further condensed by replacing the subsequence
$<<v_3\ v_4>>$ by an R-unit specifying a repetition of p3.

The permutation units are then replaced by unique
A-units after being normalized by a lexicographical ordering
of the argument subsequences. The new A-units are then
collected in a list and compared with previous A-units for
equivalence. Normalizing makes this search considerably
easier; replacing all equivalent A-units by a single one
makes the repetition detection procedure considerably faster.
When looking for repeating subsequences, the only units
that can be compared to a permutation is another such permu-
tation. For example, given $<<(P.*\ <<B>><<A>>)A>>$ where
A and B are arbitrary units, the inner subsequence $<<A>>$
cannot be viewed as a repetition with the outer unit A because
of the presence of B within the P-unit. The B must precede

- 64 -

the outer A and in particular can occur between the two

occurrences of A.   See Figure 2.4 for complete details of

these algorithms.

Figure 2.4. Permutation Construction.
===

```
CONSTRUCT-P*(SEQ): procedure;

SEQ: I-SEQUENCE;

/* CURRENT accumulates the linear sequence. */

    CURRENT := omega;

    1 <= forall i <= #SEQ;
        NN := SEQ[i].NODENUM;

/* The following assures us that unique node numbers
   will be assigned to all production names in every
   instance of a trace unit in which it occurs. */

        SEQ[i].NODENUM := NODECOUNT + NN;

/* If this node has no sons, it was preceded by nothing
   which produced input for it in the sequence and it
   is appended to no other sequence constructed.
   Notice that when a node is placed in the linear
   sequence, the SONS field (arcs) are removed. */

        if SEQ[i].SONS = omega then
            SEQ[i].SONS := omega;
            CURRENT := <SEQ[i]>;

        /* If it has only one son, then a simple
           concatenation is performed. */

        else if #SEQ[i].SONS = 1 then
            CURRENT := VISITED[s in SEQ[i].SONS];
            SEQ[i].SONS := omega;
            CURRENT := CURRENT + <SEQ[i]>;
        else

            /* a P-unit is to be constructed. */

            CURRENT := +:{ {VISITED[s]}  !
                    forall s in SEQ[i].SONS };

            /* factor out common leading subsequences. */

            FACTOR(CURRENT,PREFIX);
            SEQ[i].SONS := omega;

            /* CURRENT will become the
               SUBSEQUENCES component of the newly
               formed P-unit after repetition detection
               is applied. */

            CURRENT := +:< <REPEATS(s,SEQ[i])> !
                    forall s in  CURRENT >;
            NEWUNIT := heap(P-unit);
            NEWUNIT.SUBSEQUENCES := CURRENT;
            CURRENT := PREFIX +
                        <PSTARS(NEWUNIT)>
                            +
                        <SEQ[i]>;
        end if #SEQ[i].SONS;
```

- 66 -

```
                    end if SEQ[I].SONS = omega;

                    VISITED[NN] := CURRENT;

              end forall i:

/* The final value of CURRENT is the subsequence
   associated with the last unit of the sequence which
   is assumed to be the unique sink node of the DAG.
   Notice how extraneous nodes in the graph will auto-
   matically be lost since they would not be part of the
   sequence associated with the sink.  A node is extraneous
   if it does not contribute to the solution sequence, i.e.,
   it does not lie on any path terminating at the sink node.*/

       return CURRENT;

end CONSTRUCT-P*;


FACTOR(CURRENT,PREFIX): procedure;

CURRENT: set of I-SEQUENCE;
PREFIX: I-SEQUENCE;

       PREFIX := omega;
       TARGET := any one in CURRENT;
       CURRENT := CURRENT - { TARGET };

       repeat while(forall seq in CURRENT, seq[1] = TARGET[1]);
              PREFIX := PREFIX + <TARGET[1]>;
              TARGET := TARGET[2:];

              forall s in CURRENT;
                    s := s[2:];
              end forall s;

       end repeat;

end FACTORS;


PSTARS(PUNIT): procedure;

PUNIT: P-unit; /* newly formed. */

/* An ordering is imposed on all atoms. This ordering
   is used for normalizing P-units. The lexicographic
   sorting routine has an obvious algorithm and
   is not shown. */

       PUNIT.SUBSEQUENCES := lex-sort(PUNIT.SUBSEQUENCES)

/* FINDTRUES is a simple routine which finds the
   collection of productions which were true on enter-
   ing the production contained in the first unit of
   the permutation. It is also capable of finding the
   same for a sequence of simple units or for the
   argument sequence of an R-unit. It is used in several
   places throughout the code and is not shown explicitly.*/

       TRPS := FINDTRUES(PUNIT);

/* If PSTARNAMES is empty, then we cannot compare
   this P-unit with any other.  We call P*FIXUP which
```

```
generates new A-units. */

    if PSTARNAMES = {} then return
                            P*FIXUP(PUNIT,TRPS);
                        end if;

/* Lexico-equivalent is a Boolean function which
   compares two sequences for lexicographic equiva-
   lency.     Using lexico-equivalency allows for
   R-units. The code for lexico-equivalent is not
   shown.
    In what follows, we search PSTARNAMES for another
   P-unit which is equivalent to the one being
   formed. The canonizing makes this checking easier
   since we do not have to compare every pair of
   sequences. If one is found, both are replaced
   by a new A-unit which represents both. Notice,
   EQUIVSEQ is used to construct a 'merged'  seq-
   uence of each of the permutable subsequences.
   See the repetition routines for details of what
   EQUIVSEQ does. */

    if exists name in PSTARNAMES such that
        1 <= forall i <= #PUNIT.SUBSEQUENCES,
          lexico-equivalent(PUNIT.SUBSEQUENCES[i],
                        NAME.SUBSEQNC.SUBSEQUENCES[i])
      then
            1 <= forall j <= #PUNIT.SUBSEQUENCES;
                PUNIT.SUBSEQUENCES[j] :=
                    EQUIVSEQ(PUNIT.SUBSEQUENCES[j],
                            1,
                            NAME.SUBSEQUENCES[j],
                            1,
                            CURRENTLN(NAME.
                                    SUBSEQUENCES[j]),
                            DUMMY);
            end forall j;

            NAME.SUBSEQNC := PUNIT;
            NAME.TRUE-PS := NAME.TRUE-PS + TRPS;
            NEWNM := P*FIXUP(PUNIT,NAME.TRUE-PS);
            NEWNM.EQUIVALENTTO := NAME;
            NEWNM.ORDERING := NAME.ORDERING;

            return NEWNM;

        end if;

/* If no equivalent A-unit was found, we construct a
   new one by calling P*FIXUP. */

        return P*FIXUP(PUNIT,TRPS);

  end PSTARS;
```

- 68 -

```
P*FIXUP(PUN,TRPS): procedure;

PUN: P-unit;
TRPS: set of atoms;

/* This routine generates new atom units. */

    NNAME := heap(A-unit);
    NNAME.SUBSEQNC := PUN;
    NNAME.TRUE-PS := TRPS;
    NNAME.ORDERING := PCOUNT;
    NNAME.EQUIVALENTTO := NNAME;
    PCOUNT := PCOUNT + 1;
    PSTARNAMES := PSTARNAMES + { NNAME };

    return NNAME;

end P*FIXUP;
```

2.3.3.4 Repetition Routines

The next process is the detection of repeating subsequences and replacement of such sequences by repetition units. Unlike the permutation detection problem, which is well-defined and solved by the algorithm described above, some sequences may have several equally acceptable descriptions which use the repetition operator. In fact, the corresponding minimization problem is NP-complete in the general case (see Section 1.4.2). The algorithm that we use gives one description. It employs a helpful heuristic with a 'divide and conquer' flavor.

The sequence is scanned to identify all single occurrences of A-units or rule names in S-units. All such occurrences cannot be part of a repetition and thus divide the sequence into shorter subsequences to which the same procedure can be applied recursively. Once a candidate subsequence is found (every atom in it occurs at least twice) a left to right scan is applied repeatedly to find equivalent adjacent subsequences (beginning with the shorter) and to replace them with R-units. The procedure stops when it considers sequences whose lengths are greater than one-half the current length of the sequence. The maximum length of subsequences we have to consider decreases when repetitions are constructed.

The length of a sequence differs from the length of the tuple encoding the sequence because of the appearance of

arbitrarily complex units. For example, the length of the
following sequence as an n-tuple is 3 yet its length as
a CRAPS sequence is 5: <<A (R.* 3 <<B C D>>(W.* ( ))(U.* ())) E>>.

The *while* and *until* components of R-units are computed
from the TRUE-PS field of the S-units. A while condition
is interpreted to be all of the active productions common
to every entry of the repeating subsequence which are not
active on exiting from the subsequence. An until condition
is the opposite; everything that is active on exit but
which **was** never active previously.

The critical aspect of this procedure is the equivalence
of two sequences. Our interpretation is that the sequences
must be lexicographically equal (the same names of A-units
or production names of S-units appear in order) and aligned
on a 'unit-boundary', that is, one sequence cannot start within
an R-unit appearing at the end of the other. When two equivalent
sequences are detected, a third sequence is constructed **which**
is a merging of the R-units appearing in both.

For example,

<<A (R.* <<B>>(W.* ( )) (U.* (E))) C>>

and

<<A B (R.* <<C>> (W.* (G)) (U.* ( )))>>

are equivalent, and are merged to the sequence

<<A (R.* <<B>>(W.* ( ))(U.*(E))

(R.* <<C>> (W.* (G)) (U.* ( ))) >>.

Although merging sequences has the potential of
generating many more control statements than we might want,
we have found that the while and until conditions will
keep this to a manageable number (however, it is possible
that empty conditions are computed). In fact, whenever
a merge of two sequences occurs, we compute new while and
until conditions for any R-units that are merged, which
reflects the conditions appearing in both units. This will
mask out any erroneous iterations when the description is
used in controlling a PS. See Figure 2.5 for the details
of this entire process.

```
Figure 2.5. Repetition Detection.
===================================

REPEATS(SEQ,NXTUNIT): procedure;

SEQ: I-SEQUENCE;
NXTUNIT: I-unit; /* NXTUNIT is the unit which follows
            SEQ and which is used to calculate the
            WHILE and UNTIL conditions for any sub-
            sequences found repeating at the end of SEQ.*/

    if SEQ = omega then return omega; end if;

    NEWS := omega;
    TEST := SEQ[#SEQ];

/* Through a simple hashing scheme, the positions (ordered)
 of those units whose productions (or EQUIVALENTTO names
 in the case of A-units) appearing only once in SEQ,
 can be found easily.  We split SEQ into portions delimited
 by the unique units and recursively apply the same process
 to those portions. The FINDUNIQUES routine is not shown.*/

    UNIQUES := FINDUNIQUES(SEQ);
    if #UNIQUES = #SEQ then return SEQ; end if;
    if UNIQUES \= omega
     then
         repeat while(UNIQUES \= omega);
             FIRST := UNIQUES[1];
             UNIQUES := UNIQUES[2:];
             HOLD := SEQ[1:FIRST-1];
             FUNIT := SEQ[FIRST];
             SEQ := SEQ[FIRST+1:];
             NEWS := NEWS + REPEATS(HOLD,FUNIT)
                              + <FUNIT>;
         end repeat;

         if FUNIT = TEST then return NEWS;
         else return
                  NEWS + REPEATS(SEQ,NXTUNIT);
         end if;

    end if;

/* Everything in SEQ appears at least twice so we scan
  for repeating subsequences.  */

    for len := 1 by 1 repeat;
       if CURRENTLN(SEQ) / 2 < len then return SEQ; end if;

/* TRAIL, LEFT and RIGHT are scanning pointers. */

       TRAIL := 0;
       LEFT := 1;

/* ADVANCE positions a pointer some distance away from
  the first argument (another pointer). This distance
  is some number of units which encode a subsequence of
  length len (the second argument). The appearance of R-units
  makes this slightly difficult.  Consider the following:
```

- 73 -

< A, B, (R.* < C,A >),B,C >. When LEFT is 1 and len is 3
the pointer cannot be positioned because of the inner
R-unit. Obviously, <A,B,C> is not a repeating subsequence
because it is not aligned on a 'unit' boundary. The
third argument to ADVANCE is set to true if the positioning
problem occurs. ADVANCE returns omega as value if the
positioning is possible but the remaining length of the
sequence from that position is less than len.
 The code for ADVANCE is simple. It uses a scan of the top
level of the sequence keeping count of the cumulative
length. When the length is equal to len it returns success
of positioning but then goes on to test the remaining
length. The code for ADVANCE is not shown. */

```
    repeat while( RIGHT \= omega ) doing (
                        RIGHT := ADVANCE(LEFT,len,FLAG,SEQ));

        if FLAG then /* the positioning cannot be done*/
            TRAIL := LEFT;
            LEFT := LEFT + 1;
            continue outerloop for len = 1 by 1;
        end if;
```

/* EQUIVSEQ returns a 'merge' sequence of two
  sequences if they are equivalent. It is explained
  in the following sections of code. */

```
        MERGE := EQUIVSEQ(SEQ,LEFT,SEQ,RIGHT,len,LASTPOS);
```

/* If we found a repeating subsequence we look for
  more by calling the next routine. All variables in
  this routine are global to the following call. */

```
        if MERGE \= omega
         then
          SUB-SCAN-LOOP();

        HOLD := SEQ[LASTPOS+1:];
```

/* COLLAPSE creates a new R-unit. INTSCT, UNYUN and
  LASTPOS get values from SUB-SCAN-LOOP.
   Notice in the call to COLLAPSE a conditional expression
  appears. HOLD equal to omega implies we found a
  repeating subsequence at the end of SEQ. Therefore
  we send NXTUNIT to COLLAPSE to compute the conditions
  for the new R-unit. Otherwise we just send the first
  unit of HOLD. */

```
        NEWSEQ := COLLAPSE(MERGE,INTSCT,UNYUN,len,
                        (if HOLD=omega then NXTUNIT
                            else HOLD[1] end if) );
        if TRAIL = 0 then
            SEQ := NEWSEQ + HOLD;
            TRAIL := 1;
            LEFT := 2;
        else
            SEQ := SEQ[1:TRAIL] + NEWSEQ + HOLD;
            LEFT := TRAIL + 2;
            TRAIL :=  TRAIL + 1;
        end if TRAIL;

    else /* MERGE is equal to omega */
        TRAIL := LEFT;
        LEFT := LEFT + 1;
```

- 74 -

```
                  end if MERGE;

                  end repeat while (RIGHT;

            end for len;

      end REPEATS;


      SUB-SCAN-LOOP(): procedure;

      /* All variables within REPEATS are known here. */

            RYTE := LASTPOS + 1;

      /* INTSCT and UNYUN collect the information needed to compute
      the WHILE and UNTIL conditions. They simply accumulate the
      intersection and union, respectively, of the set of FINDTRUES
      for each instance of the repeating subsequence.
       LASTUNIT is a simple routine which returns the last unit
      appearing in a subsequence of a certain length. It is a simple
      scanning and counting algorithm which is not shown. */

            INTSCT := FINDTRUES(SEQ[LEFT]) * FINDTRUES(SEQ[RIGHT]);
            UNYUN := FINDTRUES(LASTUNIT(SEQ,LEFT,len)) +
                              FINDTRUES(LASTUNIT(SEQ,RIGHT,len));
            repeat while( MERGER \= omega )
               doing (
                     MERGER := EQUIVSEQ(MERGE,1,SEQ,RYTE,len,ATLAST));
               MERGE := MERGER;
               LASTPOS := ATLAST;
               INTSCT := INTSCT *
                              FINDTRUES(SEQ[RYTE]);
               UNYUN := UNYUN +
                              FINDTRUES(LASTUNIT(SEQ,RYTE,len));
               RYTE := LASTPOS + 1;
            end repeat;

      end SUB-SCAN-LOOP;


      CURRENTLN(SEQ): procedure;

      SEQ: I-SEQUENCE of S-, A- and R-units;

            CNTR := 0;

            1 <= forall i <= #SEQ;
               if SEQ[i] is an S-unit or A-unit then CNTR := CNTR + 1;
               else
                  CNTR := CNTR + SEQ[i].LENTH;
               end if;
            end forall;

            return CNTR;
      end CURRENTLN;
```

```
EQUIVSEQ(S1,P1,S2,P2,LEN,LAST): procedure;

S1,S2: I-SEQUENCES; /* Sequences being scanned for
                    equivalency. */
P1,P2: integer; /* pointers into S1 and S2 */
LEN: integer; /* length of subsequence being compared */
LAST: integer; /* pointer indicating last top level unit
                    in S2 scanned during the comparison. */

/* EQUIVSEQ is a recursive comparing routine. It compares
   two sequences, S1 and S2, at positions P1 and P2 of
   (CURRENTLN) length LEN and returns a merge sequence
   of the two if they were equivalent.  LAST is set to
   the last position scanned in S2 during the compare.
    The comparison procedes as follows:  Two S-units are
   equivalent if their PNAME's are equal. Two A-units
   are equivalent if they are EQUIVALENTTO the same A-
   unit. An R-unit is equivalent to some subsequence if
   its C-SEQ component is equivalent to that subsequence.
   Follow the code carefully for the R-unit comparison
   to see how the recursion works.
    The merged sequence returned is computed as follows:
   If two equivalent S- or A-units were detected, a copy
   of one of them is attached to the merged sequence. If
   an R-unit is detected to be equivalent to a subsequence,
   a copy of it is attached to the merged sequence after
   its C-SEQ component is replaced by the 'sub'-merged
   sequence produced by the recursive call to EQUIVSEQ.
   For example, given:
    <A,(R <B>), (R <C,D>)> and <A,B,(R <C>),D>, the routine
   would return the merged sequence
    <A,(R <B>),(R < (R <C>),D>) >.
    The REPLACE-W-U routine is used to merge the WHILE
   and UNTIL components of two equivalent R-units, merged
   together, by producing a DNF component computed by
   merging each of their corresponding DNF components.
   MERGEDNF does the same for simple units and is not shown.*/

     CNT := 0;
     MERGSEQ := omega;

     repeat while(CNT <= LEN);

        if P1 > #SEQ or P2 > #SEQ then return omega;
        end if;

        if SEQ[P1] is an A-unit and SEQ[P2] is an A-unit
          then
               if SEQ[P1].EQUIVALENTTO \= SEQ[P2].EQUIVALENTTO
                 then return omega;
               else
                 MERGSEQ := MERGSEQ + <SEQ[P1].EQUIVALENTTO>;
                 CNT := CNT +1;
                 LAST := P2;
                 P1 := P1 + 1;
                 P2 := P2 + 1;
               end if;

        else
          if S1[P1] is an S-unit and S2[P2] is an A-unit or
             S1[P1] is an A-unit and S2[P2] is an S-unit
            then return omega;

          else
```

```
                        if S1[P1] is an S-unit and S2[P2] is an S-unit
                          then
                            if S1[P1].PNAME \= S2[P2].PNAME
                             then return omega;
                            else
                             MERGEDNF(S1[P1],S2[P2]);
                             MERGSEQ := MERGSEQ + <S1[P1]>;
                             CNT := CNT + 1;
                             LAST := P2;
                             P1 := P1 + 1;
                             P2 := P2 + 1;
                            end if;

                        else
                          if S1[P1] is an R-unit
                          then
                              LC := S1[P1].LENTH;
                              if S2[P2] is an R-unit
                                then
                                    RC := S2[P2].LENTH;

                                  if RC = LC
                                    then
                                        HOLD := EQUIVSEQ(S1[P1].C-SEQ,
                                                         1,
                                                         S2[P2].C-SEQ,
                                                         1,
                                                         LC,
                                                         dummy);
                                      if HOLD = omega then return omega;
                                      end if;
                                      NEWUNIT := S1[P1];
                                      REPLACE-W-U(NEWUNIT,S2[P2],S2[P2]);
                                      NEWUNIT.C-SEQ := HOLD;
                                      MERGSEQ := MERGSEQ + <NEWUNIT>;
                                      CNT := CNT + LC;
                                      LAST := P2;
                                      P1 := P1 + 1;
                                      P2 := P2 + 1;

                                  else
                                    if LC > RC
                                        then
                                            HOLD := EQUIVSEQ(S1[P1].C-SEQ,
                                                             1,
                                                             S2,
                                                             P2,
                                                             LC,
                                                             LAST);
                                          if HOLD = omega then return omega;
                                          end if;
                                          NEWUNIT := S1[P1];
                                          NEWUNIT.C-SEQ := HOLD;
                                          REPLACE-W-U(NEWUNIT,
                                                      S2[P2],S2[LAST]);
                                          MERGSEQ := MERGSEQ + <NEWUNIT>;
                                          CNT := CNT + LC;
                                          P1 := P1 + 1;
                                          P2 := LAST + 1;

                                      else /* LC < RC */
                                          HOLD := EQUIVSEQ(S2[P2].C-SEQ,
                                                           1,
                                                           S1,
```

```
                                          P1,
                                          RC,
                                          TEMP);
                        if HOLD = omega then return omega;
                        end if;
                        NEWUNIT := S2[P2];
                        NEWUNIT.C-SEQ := HOLD;
                        REPLACE-W-U(NEWUNIT,
                                    S1[P1],S1[TEMP]);
                        MERGSEQ := MERGSEQ + <NEWUNIT>;
                        CNT := CNT + RC;
                        P1 := TEMP + 1;
                        LAST := P2;
                        P2 := P2 + 1;

                  end if LC > RC;

                end if LC = RC;

            else /* S2[P2] is not an R-unit */
                HOLD := EQUIVSEQ(S1[P1].C-SEQ,
                                 1,
                                 S2,
                                 P2,
                                 LC,
                                 LAST);
                if HOLD = omega then return omega;
                end if;
                NEWUNIT := S1[P1];
                NEWUNIT.C-SEQ := HOLD;
                MERGSEQ := MERGSEQ + <NEWUNIT>;
                P1 := P1 + 1;
                P2 := LAST + 1;
                CNT := CNT + LC;

            end if S2[P2] is R-unit;

        else /* S1[P1] is not an R-unit
               but S2[P2] must be an R-unit. */
            RC := S2[P2].LENTH;
            HOLD := EQUIVSEQ(S1,P1,
                             S2[P2].C-SEQ, 1,
                             RC,DUMMY);
            if HOLD = omega then return omega;
            end if;
            NEWUNIT := S2[P2];
            NEWUNIT.C-SEQ := HOLD;
            MERGSEQ := MERGSEQ + <NEWUNIT>;
            CNT := CNT + RC;
            LAST := P2;
            P2 := P2 + 1;
            P1 := P1 + RC;

        end if S1[P1] is an R-unit;

    end repeat;

    if CNT = LEN then return MERGSEQ;
    else return omega;

end EQUIVSEQ;
```

```
REPLACE-W-U(UNIT,WUNIT,UUNIT): procedure;

UNIT,UUNIT,WUINT: I-unit;

/* UNIT is an R-unit whose WHILE and UNTIL conditions
   will be merged with that of WUNIT and UUNIT. */

    UNIT.WHILE.DNF :=

        /* UNION-OR is a simple routine which merges
           two DNF components into one. It is not shown.*/

            UNION-OR(UNIT.WHILE.DNF,WUNIT.WHILE.DNF);

      if UUNIT is an R-unit then
          UNIT.UNTIL.DNF :=
              UNION-OR(UNIT.UNTIL.DNF,UUNIT.UNTIL.DNF);
      end if;

end REPLACE-W-U;

COLLAPSE(MERG,ENTER,ENDING,LEN,NXT): procedure;

MERGE: I-SEQUENCE; /* The merged sequence. */
ENTER: set of atoms; /* Rule names true on entering
                        the repeating subsequence. */
ENDING: set of atoms; /* Rule names true on leaving.*/
LEN: integer; /* Length of subsequence. */
NXT: I-unit; /* The unit which follows the repeating
                subsequence. */

/* This routine creates new R-units. */

      NEWUNIT := heap(R-unit);
      TRUEP := FINDTRUES(NXT);
      ENTER := ENTER - TRUEP;
      ENDING := TRUEP - ENDING;

/* The set of rule names is viewed as a simple conjunct. */

      NEWUNIT.LENTH := LEN;
      NEWUNIT.C-SEQ := MERGE;
      NEWUNIT.WHILE.DNF := ENTER;
      NEWUNIT.UNTIL.DNF := ENDING;

      return NEWUNIT;

end COLLAPSE;
```

2.3.3.5 Alternative Subsequence Detection

The final process is the coalescing of several descriptions of a set of solution sequences into one general description. Our construction is similar to that of Winston [49] and Hayes-Roth [19,20]. From two or more descriptions of some object we identify the points of similarity and alternate the differences.

This problem can be viewed as being equivalent to the Longest Common Subsequence problem. Since it is believed to be computationally intractable because of its NP-completeness (in the case of three or more sequences - Maier [29]), we do not intend to find the maximum number of points of similarity but rather a good heuristic approximation.

The set of descriptions is first ordered lexicographically to aid in discovering any equivalent ones which subsequently are replaced by one description. Then a target description is computed which is composed of a series of subsequences each common to all of the descriptions. This target identifies the points within a description that delimit smaller subsequences to be alternated. If the target is empty, a new target is computed. It proceeds by finding the 'best' series of subsequences common to any two descriptions. 'Best' is defined heuristically; more weight is given to A-units and R-units than S-units. If the target in this case is still empty the descriptions are collected into an ALT-unit with conditions computed as

described below.

Using the final computed target, all descriptions are
scanned for the first occurrence of the common target sub-
sequence. The initial portions of the descriptions scanned
are then collected in a set to which the entire procedure
is recursively applied. The procedure continues in this way
until the target is exhausted. A special tag unit is
appended to the end of each description to force the algo-
rithm to completion.

A difficulty arises when a target subsequence does not
appear in a particular description. In this case, a prefix
subsequence must be selected so that it will be alternated
with those prefix subsequences found in the descriptions
which matched the target subsequence. This selection process
is based on a simple heuristic: look for the first occurrence
of a single unit that is either common to the target sub-
sequence and nonmatching description, or common to one of
the subsequences extracted from a matching description and
the nonmatching description.

We demonstrate this process with an example. Given
<<B D E>>, <<B D F G>> and <<D F G >>, the following events
will occur: The target is computed to be < <<D>> <<T>> >
since D is common to all and T is the special tag unit.
Then we extract the prefixes <<B>>,<<B>> and << >>. The
entire routine is recursively applied to these. The ordering
produces <<>> and <<B>> as the new set of descriptions.

Since there is nothing common to both, the routine alter-
nates them and returns <<(A.* (<< >>( )) (<<B>> (B)))>>
to the top level. The sequence returned is concatenated
with <<D>> to produce <<(A.* (<< >> ( )) (<<B>>(B))) D>>.
Now <<T>> is the target. The prefixes collected this time
are <<E>> <<F G>> and <<F G>>. Again the recursion orders
them getting <<E>> and <<F G>>. No commonalities are
detected so we return with <<(A.* (<<E>>(E)) (<<F G>>(F G)))>>
The tag is removed from the sequence at the top level and
the final description produced is <<(A.* (<< >>( ))(<<B>>
(B))) D (A.* (<<E>>(E)) (<<F G>>(F G)))>>.

The computation for the conditions of the alternative
subsequences is very simple. For each alternate, we find
the set of rule names true on entry to each. From each of
these sets we remove any production name appearing in any
other. Intuitively, the conditions specify the productions
which are true when a particular alternative is entered but
which are not true on entering any other alternative. See
Figure 2.6 for complete details of this entire process.

After computing alternation units, all A-units are
replaced by the (recursively) cleansed versions of their
corresponding P-units. All numeric components are removed
(numbered PNAMEs and the lengths within the R-units). The
final output is the CRAPS description.

Figure 2.6. Alternation Construction.
==

```
ALTERNATIVES(DESC): procedure;

DESC: set of I-SEQUENCE; /* All compiled descriptions. */

/* ORDER lexicographically orders the descriptions and merges
   any which are equivalent. It returns a sequence of descript-
   ions. It is a simple routine using lex-sort and EQUIVSEQ
   and is not shown. */

    DE := ORDER(DESC);
    if DE = omega then return DE; end if;
    if #DE = 1 then return DE[1]; end if;
    if omega is in DE then
       DE := DE - omega;
    end if;

/* COMMON computes a subsequence common to all descriptions.*/

    TARGET := COMMON(DE);
    if TARGET = omega then
       MAX := 0;

/* PAIR-LOOP is a routine which calls the function passed
   to it as second argument for every disjoint pair of objects
   contained in its first argument. The routine passed to it
   BEST-COM-SUBSEQ, will compute the best common subsequence
   found between any two descriptions. Notice TARGET and MAX
   are global to BEST-COM-SUBSEQ.
   The code for PAIR-LOOP is not shown. */

       PAIR-LOOP(DE,BEST-COM-SUBSEQ);

       if TARGET = omega /* no commonalities at all */ then
           return COALESCE(DE); /*which forms a new ALT-unit*/
       end if;

    end if; /* A series of common subsequences was found. */

/* To force the algorithm to completion, we tag all sequences
   with a dummy A-unit. */

    TARGET := TARGET + < <dummy-A-unit> >;

    1 <= forall i <= #DE;
       DE[i] := DE[i] + <dummy-A-unit>;
    end forall i;

    FINAL := omega;

    1 <= forall i <= #TARGET;
       MATCH,NONMATCH := {};
       NULLFLAG := false;
       CLCT := omega;

/* These two variables are used globally in FIX-UPS and are
   passed to other routines.  NXTONE is used for calculating
   the DNF expressions for alternatives. */
```

```
                    NXTONE := TARGET[i+1][1];
                    SEQ := TARGET[i];

                    1 <= forall j <= #DE;
```

/* FIND-FIRST-SEQ finds the first occurrence of the second
   argument sequence in the first argument sequence. It returns
   a vector of information. The first element is the beginning
   location of the sequence that was found. The second element
   is the position in the first sequence where it ended and the
   third element is the merged sequence combining the second
   argument and the matching portion of the first argument.
   It uses a scan with repeated calls to EQUIVSEQ. The code is
   not shown. */

```
                    LOC := FIND-FIRST-SEQ(DE[j],SEQ);
                    if LOC[1] = 0 then
                        NULLFLAG := true;
                        NONMATCH := NONMATCH + {j};
                    else
                        MATCH := MATCH + {j};
                        HOLD := DE[j][1:LOC[2]];
                        DE[j] := DE[j][LOC[2]+1:];
                        CLCT := CLCT + <HOLD>;
                    end if;
                end forall j;
```

/* The following routine will handle the recursive calls to
   this routine after computing the appropriate prefixes from
   the non-matching descriptions. FINAL will be assigned values
   by FIX-UPS. */

```
                FIX-UPS();
            end forall i;
```

/* We have to remove the tag from FINAL. */

```
        FINAL := FINAL[1:#FINAL-1];
        return FINAL;
```

end ALTERNATIVES;

COMMON(DES): procedure;

DES: sequence of descriptions.

```
    MAX := maxinteger;
    DES := sort-by-length-increasing(DES);
    HLD, TARGET := DES[1];
```

/* The following loop will find a target description by
   continually removing subsequences from the target that
   are not common to all descriptions. */

```
    2 <= forall i <= #DES;
```

/* TARGET will be adjusted to hold a sequence of
   subsequences found to be common to HLD and DES[i]
   by COM-SUBSEQ. Notice that < is sent as a function
   for the comparison in the routine. The subsequence
   with the least WEIGHT is necessarily the subsequence
   common to all descriptions. See COM-SUBSEQ. */

```
                COM-SUBSEQ(HLD,DES[i],<);
                if TARGET := omega then return omega; end if;
                HLD := omega;

                1 <= forall j <= #TARGET;

                    /* The following just concatenates all
                       of the subsequences creating one sequence.*/

                    HLD := HLD + TARGET[j];

                end forall j;

            end forall i;

            return TARGET;

        end COMMON;


        BEST-COM-SUBSEQ(X,Y): procedure;

        X,Y: I-SEQUENCE;

        /* We return the common subsequence with greatest WEIGHT. */

            return COM-SUBSEQ(X,Y,>);

        end BEST-COM-SUBSEQ;

        COM-SUBSEQ(X,Y,CFUNC): procedure, value(X,Y);

        X,Y: I-SEQUENCE;
        CFUNC: Boolean function;

            INIT := 0;
            LOCALWT := 0;
            LOCALTARGET := omega;

            repeat while(X \= omega and Y \= omega);

                1 <= forall i <= #X;
                    LOC := FIND-FIRST-SEQ(Y,<X[i]>);
                    if LOC[1] \= 0 then
                        INIT := i;
                        stop forall i;
                    end if;
                end forall i;

                if INIT = 0 then return LOCALTARGET; end if;

        /* FIND-MAX-LENTH uses EQUIVSEQ to find the maximum length
           subsequence common to its two sequence arguments(first and
           third) rooted at the positions passed as second and fourth
           argument. It  is a simple scanning and counting routine
           and is not shown. */

                LEN := FIND-MAX-LENTH(X,INIT,Y,LOC[1]);
                MERGE := EQUIVSEQ(X,INIT,Y,LOC[1],LEN,LASTY);
                LOCALTARGET := LOCALTARGET + <MERGE>;

        /* WEIGHT is a heuristic weighting function which calculates
           the relative size of a sequence as follows:
```

- 85 -

Each S-unit contributes 1, each R-unit contributes
2 + the WEIGHT of its C-SEQ argument and A-units
contribute 5 each.
The routine is a simple scanning function and is not
shown. */

```
        LOCALWT := LOCALWT + WEIGHT(MERGE);
        Y = Y[LASTY+1:];

    /* ADVANCE is discussed in REPEATS above.*/

        X := X[ADVANCE(X,LOC[1]+LEN):];

    end repeat;

/* Now we use the compare function: < is used for COMMON
   and > for BEST-COM-SUBSEQ.  */

    if CFUNC(LOCALWT,MAX) then
        MAX := LOCALWT;
        TARGET := LOCALTARGET;
    end if;

end COM-SUBSEQ;


FIX-UPS(): procedure;

    HLD := omega;

/* This routine will compute the appropriate prefixes for the
   recursive call to ALTERNATIVES by calling SUB-SELECTIONS and
   then it will assign the appropriate values to FINAL.
   All variables known in ALTERNATIVES are known here.
   NULLFLAG is true if there were descriptions not matching
   with the current TARGET subsequence. */

    if NULLFLAG then
        HLD := SUB-SELECTIONS(DE,CLCT,NONMATCH,MATCH,NXTONE);
    end if;

/* If there were non empty prefixes of the descriptions
   matching the current TARGET subsequence, we alternate them
   recursively. */

    if CLCT \= omega then
        FINAL := FINAL + ALTERNATIVES({CLCT[i] !
                                        1 <= forall i <= #CLCT});
    end if;

/* If every description had a subsequence matching the current
   TARGET subsequence, we append the TARGET subsequence to FINAL.
   Otherwise, we append an ALT-unit to FINAL. This unit is an
   alternation between the leading prefixes of the non-matches
   with the current target subsequence, or, an alternation of
   the current target subsequence with the null sequence
   depending upon whether prefixes were computed or not. */

        FINAL := FINAL + (if \NULLFLAG then SEQ
                            else if HLD \= omega then
                                    COALESCE(HLD + <SEQ>)
                                else COALESCE(<SEQ,omega>)
                                end if
                            end if);
```

```
end FIX-UPS;


SUB-SELECTIONS(DE,CL,NON,MAT,NXT): procedure;

DE: set of I-SEQUENCES; /* Descriptions. */
CL: sequence of I-SEQUENCES; /* Leading prefixes of the
                                matching descriptions. */
NON: set of integers; /* Indices of the nonmatches. */
MAT: set of integers; /* Indices of the matches. */
NXT: I-unit; /* First unit of the next target subsequence. */

/* This routine computes the set of prefixes of the non-matching
   descriptions which are to be used for the recursive call to
   ALTERNATIVES. */

        HLD := omega;
        FLAG := false;
    if CL \= omega then

        forall i in NON;

            1 <= forall j <= #CL;
                UNIT := CL[j][#CL[j]];
                LOC := FIND-FIRST-SEQ(DE[i],<UNIT>);
                if LOC[1] \= 0 then
                    CL := CL + <DE[i][1:LOC[1]-1]>;
                    DE[i] := DE[i][LOC[1]:];
                    NON := NON - {i};
                    continue forall i;
                end if;
            end forall j;

        end forall i;

    end if;

    forall k in NON;

        forall j in MAT;
            UNIT := DE[j][1];
            LOC := FIND-FIRST-SEQ(DE[k],<UNIT>);
            if LOC[1] \= 0 then
                HLD := HLD + <DE[k][1:LOC[1]-2]>;
                DE[k] := DE[k][LOC[1]-1:];
                continue forall k;
            end if;
        end forall j;

        FLAG := true;

    end forall k;

/* If we couldn't compute a prefix for one of the non-matches
   we use the null prefix. */

    if FLAG then
        HLD := HLD + <omega>;
    end if;

    return HLD;

end SUB-SELECTIONS;
```

```
COALESCE(DESC): procedure;

DESC: sequence of I-SEQUENCE;

/* This routine creates new ALT-units. */

/* The conditionals for each alternative is computed first.*/

    TOTALS, SUBTOT := omega;
    CLCTRUS := omega;

    1 <= forall i <= #DESC;
       if DESC[i] = omega then
           CLCTRUS[i] := {};
       else
           CLCTRUS[i] := FINDTRUES(DESC[i][1]);
       end if;
    end forall i;

    1 <= forall i <= #CLCTRUS;

        1 <= forall j <= #CLCTRUS, j \= i;
           TOTAL[i] := TOTAL[i] + CLCTRUS[j];
        end forall j;

    end forall i;

    1 <= forall i <= #CLCTRUS;
       CLCTRUS[i] := CLCTRUS[i] - TOTAL[i];
    end forall i;

    NEWUNIT := heap(ALT-unit);

    1 <= forall i <= #DESC;
       NEWUNIT.COND[i].C-SEQ := DESC[i];
       NEWUNIT.COND[i].DNF := CLCTRUS[i];
    end forall i;

    return <NEWUNIT>;

end COALESCE;
```

- 88 -

2.3.3.6 Meta-Rule Construction

Construction of the meta-rules is based on the transition of the PS environment when firing a rule. The first type of meta-rule is concerned with how rules are activated. From the trace we can determine which rules may activate other productions when fired by   computing the set difference of the new environment with the old. Similarly, for the second type of meta-rule (how rules are inactivated), we compute the set difference of the old environment with the new. The Currently-active type meta-rule is constructed by copying the environment in which a rule was fired, and the Just-fired type is constructed from the sequence of rules fired.

All meta-rules so constructed are collected in sets MADE-ACTIVE, MADE-INACTIVE, ENVIRONS and JUST-FIREDS, respectively. They are represented in the set by a pair <L,R>, where L is a set of production names corresponding to the argument of the LHS function and R, the RHS. Details appear in Figure 2.7.

```
Figure 2.7: Meta-rule Construction.
====================================

CONSTRUCT-M-RULE(LUNIT,RUNIT): procedure;

difnewold, difoldnew: set of atoms;
pair: tuple of sets;

/* Construction of meta-rules according to the definitions.*/

     difnewold := (RUNIT.TRUE-PS + {RUNIT.PNAME}) -
                    (LUNIT.TRUE-PS + {LUNIT.PNAME});

     difoldnew := (LUNIT.TRUE-PS + {LUNIT.PNAME}) -
                    (RUNIT.TRUE-PS + {RUNIT.PNAME});

/* The first element in the pairs corresponds to the LHS of
   a meta-rule, and the second to the RHS. */

/* Type 1 meta-rule... */

     if exists pair in MADE-ACTIVE such that
        pair[1] = difnewold then
          pair[2] := pair[2] + {LUNIT.PNAME};
        else
        MADE-ACTIVE := MADE-ACTIVE +
                              {<difnewold,{LUNIT.PNAME}>};
      end if;

/* Type 2 meta-rule... */

     if exists pair in MADE-INACTIVE such that
        pair[1] = difoldnew then
          pair[2] := pair[2] + {LUNIT.PNAME};
        else
        MADE-INACTIVE := MADE-INACTIVE +
                              {<difoldnew,{LUNIT.PNAME}>};
      end if;

/* Type 3 meta-rule... */

     if exists pair in ENVIRONS such that
        pair[1] = LUNIT.TRUE-PS + {LUNIT.PNAME} then
          pair[2] := pair[2] + {LUNIT.PNAME};
        else
        ENVIRONS := ENVIRONS +
               {<LUNIT.TRUE-PS+{LUNIT.PNAME},{LUNIT.PNAME}>};
      end if;

/* Type 4 meta-rule... */

     if exists pair in JUST-FIREDS such that
        pair[1] = {LUNIT.PNAME} then
          pair[2] := pair[2] + {RUNIT.PNAME};
        else
        JUST-FIREDS := JUST-FIREDS +
                          {<{LUNIT.PNAME},{RUNIT.PNAME}>};
      end if;

end CONSTRUCT-M-RULE;
```

CHAPTER 3. EXPERIMENTS

In this chapter we discuss the experiments that were performed using the system and describe some of the insights we have gained from its behavior.

3.1 Jigsaw Puzzles

The Jigsaw Puzzle Production System (Section 1.2) consists of 33 productions containing an average of 3 data elements in the LHS, and was constructed in a few man hours. The representation was simplified so that knowledge of the physical locations of the puzzle pieces was ignored. This allowed us to concentrate on the more interesting problem of acquiring heuristics for solving problems rather than dealing with the complexities of moving objects in space.

The experiments were performed on 5 puzzles, each containing 25-30 pieces. Only one puzzle was used in the training session. The experiments showed that, on the average, 13 rules were applicable on each cycle, and with the size of the puzzles used, approximately 400 cycles were needed to solve a puzzle. In the absence of any heuristic controls, a production system such as the one in Section 1.2 would require an inordinate amount of time to search for a solution. (The reader is invited to compute the size of the search graph.) This situation appears to be an ideal one for demonstrating our approach. It seems likely that with

the introduction of simple sequencing rules, a system could
solve a problem of this type in a reasonable amount of time
and exhibit intelligent behavior. However, it does not
appear likely that brute force, heuristic search methods,
or more powerful representations would admit easy and
general solutions to the jigsaw puzzle problem.

It can be argued that the productions we have defined
can be encoded by just a few rules (perhaps 2 or 3).   In
this situation, the heuristics for solving this problem are
trivial. However, such a representation would be highly
inflexible and admit only limited or narrow behavior. We
have attempted to define a set of productions which would
allow for many solutions and many kinds of behavior.
Further, many of the productions (in particular, the pile
productions) are general enough to be used for other
substantially different problems. It would seem possible
then, that any heuristics we generate for these productions
would be applicable in many different situations, and
therefore would be of a more fundamentally general nature.

3.2 Execution with Training

The entire training session took several hours of connect time (mostly for bookkeeping) and 9 minutes of CPU time running the system on a CDC 6600 under UT LISP. The session was organized into a series of training sessions each intended to solve some subproblem of the overall problem. This allowed the human expert to experiment and decide on the most reasonable approach for demonstrating the solution of the problem. Furthermore, the limited resources available for the pattern recognition algorithms prohibited the analysis of solution sequences whose lengths were greater than 100 cycles. Each of the sequences produced from the shorter sessions were serially analyzed and the resulting descriptions concatenated to the CRAPS description contained in Appendix 2. Section 3.3 contains an abbreviated copy of the description.

The statistics produced by the PS interpreter for the complete training session is contained in Figure 3.1. Some of the statistics are concerned with the matching network produced by the PS interpreter from a compilation of the LHSs of the production memory. The output generated for this session is too large to reproduce here. However, we have included an abbreviated session of the fourth experiment running under CRAPS control in Appendix 3.

Figure 3.1 Training session statistics
==

(END -- EXPLICIT HALT)

33 PRODUCTIONS IN SYSTEM
127/300 NODES
391 PRODUCTIONS FIRED
3490 PRODUCTIONS INSTANTIATED
1808 WM TRANSACTIONS (8795 NODE ACTIVATIONS)
65200 TESTS PERFORMED
129 MAXIMUM WM SIZE
99.25831 MEAN WM SIZE
33 MAXIMUM CS SIZE
12.12788 MEAN CS SIZE
173 MAXIMUM NUMBER TOKENS STORED
113.75703 MEAN NUMBER TOKENS STORED

9.0 MINUTES CPU

3.3 Generated CRAPS Description and Meta-Rules

The CRAPS description (see Appendix 2) generated from
the trace in Appendix 1, consists of 75 concatenations and
15 repetitions; no permutations were discovered, and no
alternations were possible since only one sequence was
analyzed. The DAG which was constructed contained many
forward arcs which were correctly identified and discarded.
An abbreviated version of the description is shown in
Figure 3.2.

The fact that no permutations were generated may indi-
cate the rather limited usefulness of such a construct.
However, it appears more accurate to say that the solution
presented to the system was produced from an environment
in which parallelism was unlikely to occur. That is, the
limitations of the communication channel between man and
machine was not conducive to demonstrate parallel sequences
of actions. At any one time a limited amount of information
was displayed on the CRT screen and so the problem solution
demonstrated was forced to be more narrow or localized
decreasing the chances of parallel flows of information
(while increasing the likelihood of short repeating sub-
sequences). The problem of jigsaw puzzles would also appear
to make limited use of parallelism since most of the activity
is limited to a few puzzle pieces at any one time. However,
many problems seem to require such a construct. For example,
it is clear that an industrial robot with 2 (or more) arms
operating on an assembly line would be required to perform

certain functions in a strict sequential fashion. However, its productivity would increase significantly if its arms were controlled concurrently.

Figure 3.2 Abbreviated Puzzle Heuristics
==

    The preconditions of the simple units have been removed.

```
<<LOOK-AT-PIECE-IN-HEAP
  PICK-OBJECT-IN-VIEW
  MAKE-A-PILE
  [REPEAT (WHILE (LOOK-AT-PIECE-IN-HEAP))
          (UNTIL (EMPTY-HEAP))
    <<LOOK-AT-PIECE-IN-HEAP
      PICK-OBJECT-IN-VIEW
      PUT-OBJECT-IN-PILE>>]
  EMPTY-HEAP
  REMEMBER-CURRENT-PILE
  LOOK-AT-FIRST-IN-PILE
  REMEMBER-CURRENT-OBJECT
  LOOK-AT-NEXT-IN-PILE
  PICK-OBJECT-FROM-PILE
  FORGET-CURRENT-PILE
  MAKE-A-PILE
  FORGET-CURRENT-PILE
  PICK-A-PILE
  LOOK-AT-FIRST-IN-PILE
  [REPEAT (WHILE NIL)
          (UNTIL (PIECE-HAS-STRAIGHT-EDGE
                    REMEMBERED-OBJECT-IN-VIEW))
    <<[REPEAT (WHILE NIL)
              (UNTIL (PIECE-HAS-STRAIGHT-EDGE))
        <<LOOK-AT-NEXT-IN-PILE>>]
      [REPEAT (WHILE (PIECE-HAS-STRAIGHT-EDGE))
              (UNTIL (REMEMBERED-OBJECT-IN-VIEW))
        <<PICK-OBJECT-FROM-PILE
          FORGET-CURRENT-PILE
          PICK-A-PILE
          PUT-OBJECT-IN-PILE
          FORGET-CURRENT-PILE
          PICK-A-PILE
          LOOK-AT-FIRST-IN-PILE>>]>>]
  PICK-OBJECT-FROM-PILE
  FORGET-CURRENT-PILE
  PICK-A-PILE
  PUT-OBJECT-IN-PILE
  FORGET-REMEMBERED-OBJECT
  LOOK-AT-FIRST-IN-PILE
  PICK-OBJECT-FROM-PILE
  START-PUZZLE
  LOOK-AT-FIRST-IN-PILE
  [REPEAT (WHILE (LOOK-AT-NEXT-IN-PILE))
          (UNTIL NIL)
    <<[REPEAT (WHILE NIL)
              (UNTIL (PIECE-FITS-IN-PUZZLE))
        <<LOOK-AT-NEXT-IN-PILE>>]
      [REPEAT (WHILE (LOOK-AT-NEXT-IN-PILE
                        PIECE-FITS-IN-PUZZLE))
              (UNTIL NIL)
        <<PICK-OBJECT-FROM-PILE
          FIT-PIECE-IN-PUZZLE
          PIECE-PUT-IN-PUZZLE
          LOOK-AT-FIRST-IN-PILE>>]>>]
```

- 97 -

```
PICK-OBJECT-FROM-PILE
FIT-PIECE-IN-PUZZLE
PIECE-PUT-IN-PUZZLE
[REPEAT (WHILE NIL)
        (UNTIL (PUZZLE-IS-FINISHED))
  <<FORGET-CURRENT-PILE
    PICK-A-PILE
    LOOK-AT-FIRST-IN-PILE
    REMEMBER-CURRENT-OBJECT
    FIND-COLOR-OF-PIECE
    LOOK-AT-NEXT-IN-PILE
    [REPEAT (WHILE NIL)
            (UNTIL (REMEMBERED-OBJECT-IN-VIEW))
       <<[REPEAT (WHILE NIL)
                 (UNTIL (PIECE-HAS-CURRENT-COLOR))
            <<LOOK-AT-NEXT-IN-PILE>>]
         [REPEAT (WHILE (FORGET-COLOR-OF-PIECE
                         PIECE-HAS-CURRENT-COLOR))
                 (UNTIL (REMEMBERED-OBJECT-IN-VIEW))
            <<PICK-OBJECT-FROM-PILE
              FORGET-CURRENT-PILE
              PICK-A-PILE
              PUT-OBJECT-IN-PILE
              FORGET-CURRENT-PILE
              PICK-A-PILE
              LOOK-AT-FIRST-IN-PILE>>]>>]
    FORGET-REMEMBERED-OBJECT
    PICK-OBJECT-FROM-PILE
    FORGET-CURRENT-PILE
    PICK-A-PILE
    PUT-OBJECT-IN-PILE
    [REPEAT (WHILE LOOK-AT-FIRST-IN-PILE)
            (UNTIL (PILE-IS-EMPTY))
       <<LOOK-AT-FIRST-IN-PILE
         PICK-OBJECT-FROM-PILE
         FIT-PIECE-IN-PUZZLE
         PIECE-PUT-IN-PUZZLE>>]>>]
FORGET-CURRENT-PILE
[REPEAT (WHILE (PICK-A-PILE)
               FORGET-REMEMBERED-PILE))
        (UNTIL (THERE-ARE-NO-PILES))
  <<PICK-A-PILE
    DESTROY-A-PILE>>]
PUZZLE-IS-FINISHED>>
```

- 98 -

It is interesting to see exactly what heuristics are
encoded by this description. The first 5 units:

```
<LOOK-AT-PIECE-IN-HEAP
 PICK-OBJECT-IN-VIEW
 MAKE-A-PILE
 [REPEAT (WHILE (LOOK-AT-PIECE-IN-HEAP))
                (UNTIL (EMPTY-HEAP))
     <LOOK-AT-PIECE-IN-HEAP
      PICK-OBJECT-IN-VIEW
      PUT-OBJECT-IN-PILE>]
 EMPTY-HEAP
```

have the effect of ordering the pieces in the heap. It does
so by picking (arbitrarily) a piece from the heap, putting
it in a pile (MAKE-A-PILE), and repeating the same for the
remaining pieces. Since the piles are in fact queues, the
first piece placed in the pile will be the first piece to
be viewed from the pile. The initial arbitrary selection
of a piece in the heap is a source of error as we shall
see shortly.

The next part of the sequence,

```
REMEMBER-CURRENT-PILE
LOOK-AT-FIRST-IN-PILE
REMEMBER-CURRENT-OBJECT
LOOK-AT-NEXT-IN-PILE
```

has the effect of initializing the pile for an ordered search.
The first piece in the pile is tagged by REMEMBER-CURRENT-
OBJECT and then placed at the end of the pile by
LOOK-AT-NEXT-IN-PILE so that when it comes into view later,
we will know that we have scanned the entire pile.
This is the point at which an error is introduced by the
heuristics specified in the initial segment. The actual

- 99 -

preconditions of REMEMBER-CURRENT-OBJECT (see Appendix 2)

specifies that PIECE-HAS-STRAIGHT-EDGE, that is, the first

object in the pile, which is the first piece picked from

the heap, must have a straight edge. Therefore, the initial

selection of a piece from the heap should not be arbitrary,

but instead should be one with a straight edge. This

problem could have been avoided with additional training

sequences for the initial subsequence and using the alterna-

tion routine to specify alternative actions. These errors

are handled by the meta-rule mechanism.

The next part of the description,

```
PICK-OBJECT-FROM-PILE
FORGET-CURRENT-PILE
MAKE-A-PILE
FORGET-CURRENT-PILE
PICK-A-PILE
LOOK-AT-FIRST-IN-PILE
[REPEAT (WHILE NIL)
        (UNTIL (PIECE-HAS-STRAIGHT-EDGE
                REMEMBERED-OBJECT-IN-VIEW))
   <[REPEAT (WHILE NIL)
            (UNTIL (PIECE-HAS-STRAIGHT-EDGE))
       <LOOK-AT-NEXT-IN-PILE>]
    [REPEAT (WHILE (PIECE-HAS-STRIGHT-EDGE))
                   (UNTIL (REMEMBERED-OBJECT-IN-VIEW))
       <PICK-OBJECT-FROM-PILE
        FORGET-CURRENT-PILE
        PICK-A-PILE
        PUT-OBJECT-IN-PILE
        FORGET-CURRENT-PILE
        PICK-A-PILE
        LOOK-AT-FIRST-IN-PILE>]>]
PICK-OBJECT-FROM-PILE
FORGET-CURRENT-PILE
PICK-A-PILE
PUT-OBJECT-IN-PILE
FORGET-REMEMBERED-OBJECT
```

has the effect of creating a second pile composed of all of

the pieces with straight edges. The initial pile is

repeatedly scanned (until the tagged piece is seen) and
for each piece in view which has a straight edge, the piece
is removed from the first pile and placed in the second.
The appropriate conditions are encoded by the while and
until components of the repetition units.

   We now have one pile of indistinguished pieces and a
second composed of all the straight edge pieces. We can
now begin the puzzle by building the outside edges first
from the pieces in the second pile until it is exhausted.
The following subsequence performs these actions.

```
     LOOK-AT-FIRST-IN-PILE
     PICK-OBJECT-FROM-PILE
     START-PUZZLE
     LOOK-AT-FIRST-IN-PILE
     [REPEAT (WHILE (LOOK-AT-NEXT-IN-PILE))
             (UNTIL NIL)
        <[REPEAT (WHILE NIL)
                 (UNTIL (PIECE-FITS-IN-PUZZLE)
            <LOOK-AT-NEXT-IN-PILE>]
         [REPEAT (WHILE (LOOK-AT-NEXT-IN-PILE
                          PIECE-FITS-IN-PUZZLE))
                 (UNTIL NIL)
            <PICK OBJECT-FROM-PILE
             FIT-PIECE-IN-PUZZLE
             PIECE-PUT-IN-PUZZLE

             LOOK-AT-FIRST-IN-PILE>]>]
     PICK-OBJECT-FROM-PILE
     FIT-PIECE-IN-PUZZLE
     PIECE-PUT-IN-PUZZLE
```

   Attention is now turned to the first pile. The following
rather lengthy procedure is repeated until the first pile is
exhausted in which case all of the puzzle pieces will have
been placed in the puzzle.

   If you study a children's puzzle, you will observe
that (usually) all of the pieces with the same average

color are connected, and further, each color group is connected to the edge of the puzzle. This suggests the following procedure which was used by the trainer. We look at the first piece in the first pile, note its color, tag it and place it at the end of the pile. Then we scan the entire pile, removing those pieces with the same average color and place them in a separate pile, called the color pile. This procedure is repeated until we encounter the tagged piece, which is removed and placed in the color pile. We then exhaustively scan the color pile, selecting the pieces that fit in the puzzle and placing them in the puzzle. The connectivity property of children's puzzles insures that this pile will be exhausted and no infinite loop through the pile is possible. Once this is accomplished, attention is turned back to the original pile, another color is selected and the entire process is repeated until the puzzle is finished. The following subsequence encodes the above heuristics.

```
[REPEAT (WHILE NIL)
        (UNTIL (PUZZLE-IS-FINISHED))
   <FORGET-CURRENT-PILE
    PICK-A-PILE
    LOOK-AT-FIRST-IN-PILE
    REMEMBER-CURRENT-OBJECT
    FIND-COLOR-OF-PIECE
    LOOK-AT-NEXT-IN-PILE
    [REPEAT (WHILE NIL)
            (UNTIL (REMEMBERED-OBJECT-IN-VIEW))
       <[REPEAT (WHILE NIL)
                (UNTIL (PIECE-HAS-CURRENT-COLOR))
           <LOOK-AT-NEXT-IN-PILE>]
        [REPEAT (WHILE (FORGET-COLOR-OF-PIECE
                              PIECE-HAS-CURRENT-COLOR))
                (UNTIL (REMEMBERED-OBJECT-IN-VIEW))
           <PICK-OBJECT-FROM-PILE
            FORGET-CURRENT-PILE
            PICK-A-PILE
            PUT-OBJECT-IN-PILE
            FORGET-CURRENT-PILE
            PICK-A-PILE
            LOOK-AT-FIRST-IN-PILE>]>]
    FORGET-REMEMBERED-OBJECT
    PICK-OBJECT-FROM-PILE
    FORGET-CURRENT-PILE
    PICK-A-PILE
    PUT-OBJECT-IN-PILE
    [REPEAT (WHILE (LOOK-AT-FIRST-IN-PILE))
            (UNTIL (PILE-IS-EMPTY))
       <LOOK-AT-FIRST-IN-PILE
        PICK-OBJECT-FROM-PILE
        FIT-PIECE-IN-PUZZLE
        PIECE-PUT-IN-PUZZLE>]>]
```

Now that all of the piles are exhausted and all of
the pieces are in the puzzle, all of the piles are scanned
in turn, destroying each of them until none remain :

```
    FORGET-CURRENT-PILE
    [REPEAT (WHILE (PICK-A-PILE
                        FORGET-REMEMBERED-PILE))
            (UNTIL (THERE-ARE-NO-PILES))
       <PICK-A-PILE
        DESTROY-A-PILE>]
    PUZZLE-IS-FINISHED>
```

Now the puzzle is finished. Appendix 3 contains an
experiment using the above heuristics.

The analysis of the trace produced from the training session also produced 192 meta-rules: 47, 35, 90 and 20 of types 1, 2, 3 and 4 respectively.   There are too many meta-rules to be reproduced here, but an arbitrary selection of a few of each type are shown in Figure 3.3.

```
Figure 3.3 Samples of the generated Meta-rules
===============================================

Type 1 Meta-rules:

[(<WANT-ACTIVE>
              (THERE-ARE-NO-PILES)
                    ) -->
                         (<TRY-TO-FIRE> (DESTROY-A-PILE))]
[(<WANT-ACTIVE>
              (FORGET-CURRENT-PILE
               PILE-IS-EMPTY
               DESTROY-A-PILE
               REMEMBER-CURRENT-PILE)
                    ) -->
                         (<TRY-TO-FIRE> (PICK-A-PILE))]

Type 2 Meta-rules:

[(<WANT-INACTIVE>
              (FORGET-REMEMBERED-PILE)
                    ) -->
                         (<TRY-TO-FIRE> (FORGET-REMEMBERED-PILE))]
[(<WANT-INACTIVE>
              (CLOSE-EYES
               OBJECT-IN-HAND-IN-VIEW
               FIND-COLOR-OF-PIECE
               FORGET-COLOR-OF-PIECE
               PIECE-HAS-CURRENT-COLOR
               PIECE-FITS-IN-PUZZLE
               FIT-PIECE-IN-PUZZLE
               PIECE-PUT-IN-PUZZLE
               PUT-OBJECT-IN-PILE
               REMEMBER-CURRENT-OBJECT
               REMEMBERED-OBJECT-IN-VIEW
               REMEMBERED-OBJECT-IN-HAND)
                    ) -->
                         (<TRY-TO-FIRE> (PIECE-PUT-IN-PUZZLE))]

Type 3 Meta-rules:

[(<CURRENTLY-ACTIVE>
              (PUZZLE-IS-FINISHED
               THERE-ARE-NO-PILES)
                    ) -->
                         (<TRY-TO-FIRE> (PUZZLE-IS-FINISHED))]
[(<CURRENTLY-ACTIVE>
              (PUZZLE-IS-FINISHED
               THERE-ARE-NO-PILES
               FORGET-REMEMBERED-PILE)
                    ) -->
                         (<TRY-TO-FIRE>
                                   (FORGET-REMEMBERED-PILE))]

Type 4 Meta-rules:

[(<JUST-FIRED> (LOOK-AT-PIECE-IN-HEAP))
                         -->
                         (<TRY-TO-FIRE>
                                   (PICK-UP-OBJECT-IN-VIEW))]
[(<JUST-FIRED> (PICK-UP-OBJECT-IN-VIEW))
                         -->
                         (<TRY-TO-FIRE>
                                   (PUT-OBJECT-IN-PILE
                                    MAKE-A-PILE))]
```

The heuristics specified in the preceding pages are not general enough to solve every jigsaw puzzle. The human expert made certain assumptions that were reflected in the CRAPS description. In particular, when presented with a puzzle which does not have the connectivity property, it is possible that the system would fail to find a solution. However, the description is biased towards the solution demonstrated by the expert and so would exhibit intelligent behavior when presented with the appropriate inputs. A more **general** solution would be possible by conducting more training sessions, perhaps with different trainers, and using the alternation operator or the meta-rule mechanism. The meta-rules encode reasonable heuristics and correct statements of control in most cases. They are very different from those described by Davis and Kibler whose formalisms are highly data-dependent. Davis' meta-rules are acquired explicitly from a human expert and are composed of predicates applied to the data in WM. Kibler's meta-rules specify in effect 'data-flow links' between rules, and are compiled statically from production memory by an analysis of the contents of the LHS and RHS of rules. The rules we have constructed are dependent only on the conflict set of rules and the dynamic behavior of the program. This has the effect of limiting the suggested rules to those that were used in a solution. A large number of such rules are produced because of the large number of possible conflict sets contained in the solution sequence.

The meta-rules that have been defined here are flawed
by their limited scope or context. The LHS of the meta-
rules interrogate only the current conflict set or the previ-
ously fired production. In several similar situations,
the rules MAKE-A-PILE and PICK-A-PILE are suggested with
equal weight because of this limited scope. In this case,
an arbitrary selection is not adequate and a deeper analysis
of the situation is required to make the correct choice.
More context within the solution sequence should be used
to disambiguate the situation. We view our meta-rule
construct as a starting point from which to analyze and
understand the use of this type of mechanism. It is our
belief that such rules exhibit adequate behavior when used
in conjunction with the CRAPS descriptions. The primitives
defined were derived from insights gained from several
preliminary experiments with CRAPS. In view of the perform-
ance of the jigsaw puzzle PS, more research into the
automatic construction of meta-rules is highly desirable.
These points are discussed further in the following sections.

3.4 Execution with Heuristics

The original puzzle was easily solved under CRAPS control in 9.5 minutes with no meta-rule calls. The additional time required was due to the CRAPS implementation within the interpreter.

A second quite different puzzle was solved in 10.5 minutes but required 5 meta-rule calls. The situation described in the previous section (the first piece picked from the heap is required to have a straight edge) occurred and was correctly handled by the meta-rules. The first piece in the pile, although it did not have a straight edge, was tagged and eventually placed in the pile with the straight edge pieces. As the outside edges of the puzzle were being built, when the tagged piece was encountered and found to fit into the puzzle, the meta-rules suggested that it be placed in the puzzle. Subsequently, the CRAPS description worked correctly.

The following statistics were produced for this experiment.

```
(END -- EXPLICIT HALT)
(5 -- META RULE CALLS)
33 PRODUCTIONS IN SYSTEM
127/300 NODES
416 PRODUCTIONS-FIRED
3776 PRODUCTIONS-INSTANTIATED
1880 WM-TRANSACTIONS (70605 NODE ACTIVATIONS)
(69801 TESTS PERFORMED)
(137 MAXIMUM WM SIZE)
(107.32692 MEAN WH SIZE)
(35 MAXIMUM CS SIZE)
(13.12740 MEAN CS SIZE)
(181 MAXIMUM NUMBER TOKENS STORED)
(117.66587 MEAN NUMBER TOKENS STORED)

10.5 MINUTES CPU
```

The third puzzle failed to be solved because of an incorrect choice specified by the meta-rules. After 193 cycles, it had succeeded in ordering the heap and building the pile of straight edge pieces. The same situation occurred as described in the second experiment except that this time the tagged element did not fit in the puzzle when the outside edges were being built. At that point, the system looped through the pile containing the single tagged piece (all of the straight edge pieces had been placed in the puzzle). The repetition unit, which repeated LOOK-AT-NEXT-IN-PILE with the condition until PIECE-FITS-IN-PUZZLE never terminated.

The fourth puzzle worked properly. As you will note in Appendix 3, a very similar situation occurred as that described above. This time, the second piece in the pile, which is the focus of attention of the PICK-OBJECT-FROM-PILE unit following the REMEMBER-CURRENT-OBJECT unit, is also required by the preconditions to have a straight edge. In this case it did not. Rather than placing this piece in the pile of straight edge pieces, the meta-rules suggested the LOOK-AT-NEXT-IN-PILE production. This led to the correct choice of a piece from the pile (which had a straight edge) and subsequently the CRAPS description worked correctly.

A fifth puzzle was run without the CRAPS description and with the meta-rules alone. After 73 cycles and approximately 2.5 minutes, it had successfully ordered the heap by

piling the pieces. Immediately afterwards, the rules attempted to build a pile of straight edge pieces. At this time, both MAKE-A-PILE and PICK-A-PILE became equally likely candidates. The wrong choice was made, and the program proceeded to arbitrarily make piles in an almost hysterical fashion.

CHAPTER 4. CONCLUSIONS

4.1 Major Results

It is believed by many researchers that an important quality of intelligent behavior is the ability to improve performance with experience, and that generalizing a concept is a critical aspect of learning. In CRAPS, a form of generalization occurs when a repeating subsequence is collapsed into a repetition unit. Although quite restricted in scope, CRAPS shows how a system might learn procedures, a form of learning which is believed to be very important. Although there are a number of very difficult technical problems, it seems to us that with more powerful pattern recognition techniques and more powerful generalizations of control statements, this approach could be very fruitful.

With less ambitious designs, CRAPS can be viewed as a programming aid for the designer and implementer of a large AI problem-solving knowledge base. One of the most error prone and difficult tasks in the development of an AI problem-solving system is the fine-tuning of the system with heuristic controls to minimize search times through a large data base of facts. The CRAPS approach might be useful in fine-tuning a declarative knowledge base as opposed to 'hand-compiling' control elements to effect competent performance in such a system.

The power of the descriptions produced is limited by both the expressive power of the CRAPS primitives, and the level of sophistication of the pattern recognition algorithms that have been developed. For example, during repetition detection no notion of similar subsequence is used; furthermore, it is not possible for alternation to appear within repetitions. Despite this, the algorithms are powerful enough to detect interesting patterns and subsequently interesting heuristics.

Our experiments suggest that this approach will have the best chance of success when the encoding of the knowledge of the problem domain is such that on any execution cycle a small number of productions and only one instantiation of each production is applicable. For example, the initial segment of the CRAPS description for the jigsaw puzzle problem (see Appendix 2) specifies repeatedly picking a piece from the heap and placing it in a pile. In actuality, the number of instantiations of the LOOK-AT-PIECE-IN-HEAP production during this cycle is equal to the number of pieces currently in the heap. The CRAPS description does not specify which instantiation to choose; but only the name of the production to choose. This affects the subsequent ordering of the pieces in the pile, and possibly the preconditions of the units which follow in the description. This is demonstrated by the REMEMBER-CURRENT-OBJECT simple unit following the EMPTY-HEAP simple unit.

The preconditions for the unit specifies that the piece to remember (the first in the pile) should have a straight edge. But this depends on the first piece which was picked up from the heap, that is, the instantiation of PICK-UP-OBJECT-IN-VIEW that was selected. This suggests one way in which CRAPS could evolve: the execution traces should include not only productions but also instantiations of productions. Furthermore, the initial segmentation of the trace sequence into subsequences (or subproblems) specified by the human expert is not explicit in the final CRAPS description, and therefore contextual or goal information should be included. However, recognizing patterns in such sequences is a much more difficult problem. The approach outlined in this thesis is viewed as an initial step in the understanding of this more general problem.

The meta-rule construct we have defined seems quite effective. Although limited in scope, the meta-rules contain quite accurate and useful control information. The suggestion of a single rule in the RHS of a meta-rule effects a limited or local modification. More substantial statements of control can be effected by allowing arbitrary CRAPS descriptions in the RHS. However, a correspondingly more sophisticated analysis of both the solution sequence and CRAPS description would be required.

4.2 Future Research

There are many directions in which the present work can proceed. Pattern analysis is a long standing problem which obviously should be a continued research endeavor. The techniques outlined in this thesis leave much room for improvement and extension. The control language should be redefined to allow more powerful control primitives containing contextual or goal/subgoal information. An interesting question is the effect parallel machine architectures could have on the kind of analysis performed in this thesis.

The meta-rule construct requires more research in allowing for more powerful statements, such as CRAPS descriptions in the RHS and perhaps data-dependent conditions in the LHS. The implementation of the meta-rules, and in particular the selection algorithm for 'suggested' rules can be improved significantly by allowing for backtracking and other ordered search regimes.

With more powerful meta-rules, the correctness of the complete CRAPS description would be less critical; we might then hope to get successful performance with meta-rules alone, as suggested by the partial success on the fifth puzzle. However, this would probably require more understanding of some difficult context and control problems.

The existing system could be extended along the lines alluded to earlier. The utility and importance of both

counter-examples and data sequences (providing more specific knowledge about the problem domain) has been noticed by many researchers. The CRAPS system could be extended to include both types of information. Further, the existing system does not include the human engineering aspects of the type illustrated by Davis' system, and should be extended in this direction.

In Angluin [2,4], it has been shown that the pattern recognition problem attempted in this thesis is NP-complete. We have attempted to provide a good heuristic approximation for these problems. A theoretical basis of the feasibility of a more general approach is obviously needed. For instance, what is the necessary power (i.e. context free, regular expressions) of both the control language and the underlying knowledge representation model to effect competent performance in ill-defined problem domains (including problems for which no algorithmic procedure is known, such a computer vision).

In conclusion, the scope of the problems outlined in this thesis are of paramount importance to the field of artificial intelligence. Any small step made in solving the more general problems of course is significant for the entire field. We do not claim to have provided a solution, but we have identified a starting point, and a direction in which to proceed towards the solution of this enormously complex problem.

Bibliography

[1]   Abrahams, P., "Application of LISP to Sequence Pre-
      diction," Information International Report, 1965.

[2]   Angluin, D., "An Application of the Theory of Compu-
      tational Complexity to the Study of Inductive
      Inference," Ph.D. Thesis, Dept. of E E. and Comp.
      Sci., U. Calif., Berkeley, 1976.

[3]   Angluin, D., "Finding Patterns Common to a Set of
      Strings," SIGACT Proc. Symp. on Theory of Comp.,
      1979.

[4]   Angluin, D., "On the Complexity of Minimum Inference
      of Regular Sets," unpublished manuscript, 1977.

[5]   Biermann, A., "On the Inference of Turing Machines
      from Sample Computations," Artif. Intell., 3, 1975.

[6]   Chaitin, G., "A Theory of Program Size Formally
      Identical to Information Theory," J. ACM, 22, 1975.

[7]   Cohn, A., "High Level Proof in LCF," Proc. Fourth
      Workshop on Automated Deduction, Texas, 1979.

[8]   Davis, R., and King, J., "An Overview of Production
      Systems," Stanford U., AI Lab Memo, AIM-271, 1975.

[9]   Davis, R., "Applications of Meta Level Knowledge to
      the Construction, Maintenance and Use of Large
      Knowledge Bases," Ph.D. Thesis, Stanford U., 1976.

[10] Deliyanni, A., and Kowalski, R., "Logic and Semantic
      Networks," CACM, 22-23, 1979.

[11] Dewar, R., The SETL Programming Language, Courant
        Institute, NYU, 1978.

[12] Evans, T., "A Program for the Solution of a Class of
        Geometric-Analogy Intelligence Test Questions,"
        Semantic Information Processing, M. Minsky (ed.),
        1968.

[13] Feldman, J., "First Thoughts on Grammatical
        Inference," Stanford U., AI Lab Memo 55, 1967.

[14] Fikes, R., Hart, P., and Nilsson, N., "Learning and
        Executing Generalized Robot Plans," Artif. Intell.,
        3, 1972.

[15] Forgy, C., and McDermott, J., "The OPS2 Reference
        Manual," Carnegie-Mellon U., Dept. Comp. Sci., 1977.

[16] Fosdick, L., and Osterweil, L., "Data Flow Analysis
        in Software Reliability," Computing Surveys, 8-3,
        1976.

[17] Garey, M., and Johnson, D., Computers and Intractability,
        A Guide to the Theory of NP-Completeness, W. H.
        Freeman and Co., 1979.

[18] Habermann, A., "Path Expressions," Carnegie-Mellon U.,
        Dept. Comp. Sci., 1975.

[19] Hayes-Roth, F., and McDermott, J., "An Interference
        Matching Technique for Inductive Abstractions,"
        CACM, 21-5, 1978.

[20] Hayes-Roth, F., and McDermott, J., "Knowledge Acquisi-
        tion from Structural Descriptions," Proc. IJCAI, 5,
        1977.

[21] Hedrick, C., "Learning Production Systems from
      Examples," Artif. Intell., 7, 1976.

[22] Hewitt, C., "Description and Theoretical Analysis
      (Using Schemata) of PLANNER: A Language for
      Proving Theorems and Manipulating Models in a Robot,"
      MIT, AI Lab, TR-258, 1972.

[23] Hopcroft, E., and Ullman, J., Formal Languages and
      Their Relation to Automata, Addison-Wesley, 1969.

[24] Hunt, Marin, and Stone, Experiments in Induction,
      Academic Press, 1966.

[25] Kibler, D., "Power, Efficiency, and Correctness of
      Transformation Systems," Ph.D. Thesis, Comp. Sci.
      Dept., U. of Calif., Irvine, 1978.

[26] Kolmogorov, A., "Three Approaches to the Quantitative
      Definition of Information," Problems in Information
      Transmission, 1, 1965.

[27] Knuth, D., The Art of Computer Programming, Sorting
      and Searching, Vol. 3, Addison-Wesley, 1969.

[28] Lee, S., Gerhart, S., and de Roever, W., "The Evolu-
      tion of List-Copying Algorithms," Proc. Principles
      of Programming Languages, 6, 1979.

[29] Maier, D., "The Complexity of Some Problems on Sub-
      sequences and Supersequences," J. ACM, 24-2, 1978.

[30] McDermott, J., and Forgy, C., "Production System
      Conflict Resolution Strategies," Carnegie-Mellon
      U., Dept. Comp. Sci., 1976.

[31] Meltzer, B., "The Semantics of Induction and the Possibility of Complete Systems of Inductive Inference," Artif. Intell., 1, 1970.

[32] Newell, A., "Production Systems: Models of Control Structures," in Visual Information Processing, W. Chase (ed.), Academic Press, 1973.

[33] Phillips, J., "Program Inference from Traces Using Multiple Knowledge Sources," Proc. IJCAI, 5, 1977.

[34] Plotkin, G., "A Note on Inductive Generalization," Mach. Intell., 5, 1970.

[35] Popplestone, R., "An Experiment in Automatic Induction," Mach. Intell., 5, 1970.

[36] Rychener, M., "Control Requirements for the Design of Production System Architectures," SIGPLAN Notices, 12-8, 1977.

[37] Rychener, M., "Production Systems as a Programming Language for Artificial Intelligence Research," Ph.D. Thesis, Carnegie-Mellon U., Dept. Comp. Sci., 1976.

[38] Schmidt, C. Sridharan, N., and Goodson, J., "The Plan Recognition Problem: An Intersection of Psychology and Artificial Intelligence," Artif. Intell., 11, 1978.

[39] Sickel, S., "A Search Technique for Clause Inter-connectivity Graphs," IEEE Trans. on Computers, Special Issue on Autom. Theorem Proving, 1976.

[40] Solomonoff, R., "A Formal Theory of Inductive
     Inference," Information and Control, 7, 1964.

[41] Sperling, M., "Inferential Learning Through Counter-
     example Construction," Ph.D. Thesis, New York U.,
     Comp. Sci. Dept., 1976.

[42] Summers, P., "Program Construction from Examples,"
     IBM Report 24345, 1975.

[43] Tarjan, R., "Depth-First Search and Linear Graph
     Algorithms," SIAM J. Comput., 2, 1972.

[44] Tarjan, R., "Finding Dominators in Directed Graphs,"
     SIAM J. Comput., 3, 1974.

[45] Tarjan, R., "Testing Flow Graph Reducibility," J.
     Computer and System Sci., 9, 1974.

[46] Waldinger, R., and Lee, R., "PROW: A Step Toward
     Automatic Program Writing," Proc. IJCAI, 1969.

[47] Waterman, D., "Generalization Learning Techniques for
     Automating the Learning of Heuristics," Artif.
     Intell., 1, 1970.

[48] Winograd, T., "Frame Representation and the
     Declarative/Procedural Controversy," in Representa-
     tion and Understanding, Bobrow and Collins (eds.),
     Academic Press, 1975.

[49] Winston, P., "Learning Structural Descriptions from
     Examples," MIT, MAC Technical Report-76, 1976.

APPENDIX 1
==========

ABBREVIATED TRACE FOR PUZZLE PROBLEM.

THE SEQUENCE HAS BEEN CONDENSED. THE HUMAN
EXPERT DIVIDED THE TRAINING SESSION INTO SEVERAL
SESSIONS AS INDICATED BELOW. THE SUBSEQUENCES
WERE SERIALLY ANALYZED, AND THE RESULTING (SUB)
DESCRIPTIONS WERE CONCATENATED. ONE FINAL PASS OF
THE REPETITION DETECTION ALGORITHM WAS APPLIED TO THIS
SEQUENCE PRODUCING THE FINAL DESCRIPTION IN APPENDIX 2.

PRODUCTION NAME ABBREVIATION TABLE

LOOK-AT-PIECE-IN-HEAP	L-A-P-I-H
LOOK-AT-OBJECT-IN-HAND	L-A-O-I-H
CLOSE-EYES	C-E
OBJECT-IN-HAND-IN-VIEW	O-I-H-I-V
PICK-UP-OBJECT-IN-VIEW	P-U-O-I-V
PUT-PIECE-DOWN-IN-HEAP	P-P-D-I-H
EMPTY-HEAP	E-H
FIND-COLOR-OF-PIECE	F-C-O-P
FORGET-COLOR-OF-PIECE	FT-C-O-P
PIECE-HAS-STRAIGHT-EDGE	P-H-S-E
PIECE-HAS-CURRENT-COLOR	P-H-C-C
START-PUZZLE	S-P
PIECE-FITS-IN-PUZZLE	P-F-I-P
FIT-PIECE-IN-PUZZLE	F-P-I-P
PIECE-PUT-IN-PUZZLE	P-P-I-P
PUZZLE-IS-FINISHED	P-I-F
MAKE-A-FILE	M-A-F
PICK-A-FILE	P-A-F
PICK-OBJECT-FROM-FILE	P-O-F-F
PUT-OBJECT-IN-FILE	P-O-I-F
FORGET-CURRENT-FILE	F-C-F
FILE-IS-EMPTY	F-I-E
DESTROY-A-FILE	D-A-F
THERE-ARE-NO-FILES	T-A-N-F
LOOK-AT-FIRST-IN-FILE	L-A-F-I-F
LOOK-AT-NEXT-IN-FILE	L-A-N-I-F
REMEMBER-CURRENT-FILE	R-C-F
REMEMBERED-FILE-IS-CURRENT	R-F-I-C
FORGET-REMEMBERED-FILE	F-R-F
REMEMBER-CURRENT-OBJECT	R-C-O
FORGET-REMEMBERED-OBJECT	F-R-O
REMEMBERED-OBJECT-IN-VIEW	R-O-I-V
REMEMBERED-OBJECT-IN-HAND	R-O-I-H

TRACE SEQUENCE FOR PUZZLE PROBLEM
(PRODUCTION NAMES ARE ABBREVIATED)

EMPTY THE HEAP AND FILE THE PIECES:

<<<<<
    (L-A-P-I-H
        [L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H
         L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H
         L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H
         L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H
         T-A-N-P]
        (8 133)   (138 139))
    (P-U-O-I-V
        [L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H
         L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H
         L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H
         L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H
         R-C-O F-C-O-P C-E T-A-N-P]
        (137 139)   (140 145))
    (M-A-P
        [P-P-D-I-H S-P O-I-H-I-V L-A-P-I-H L-A-P-I-H L-A-P-I-H
         L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H
         L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H
         L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H
         L-A-P-I-H L-A-P-I-H L-A-P-I-H R-C-O F-C-O-P C-E
         T-A-N-P]
        (143 144 145)   (146 155))
    (L-A-P-I-H
        [P-O-F-P L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H
         L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H
         L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H
         L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H
         R-C-O F-C-O-P C-E D-A-P F-C-P R-C-P]
        (150 151 152 153 154)   (156 157))
    (P-U-O-I-V
        [L-A-F-I-P P-H-S-E L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H
         L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H
         L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H
         L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H
         L-A-P-I-H R-C-O F-C-O-P C-E D-A-P F-C-P
         R-C-P]
        (151 152 157)   (158 163))
    (P-O-I-P
        [P-P-D-I-H S-P O-I-H-I-V L-A-F-I-P P-H-S-E L-A-P-I-H
         L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H
         L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H
         L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H
         L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H R-C-O F-C-O-P
         C-E D-A-P F-C-P R-C-P]
        (161 162 163)   (164 168))
    (L-A-P-I-H
        [L-A-F-I-P P-H-S-E L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H
         L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H
         L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H
         L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H L-A-P-I-H
         R-C-O F-C-O-P C-E D-A-P F-C-P R-C-P]
        (152 166 168)   (169 170))
    (P-U-O-I-V

```
       [L-A-F-I-P L-A-F-I-H L-A-F-I-H L-A-F-I-H L-A-F-I-H L-A-F-I-H
        L-A-F-I-H L-A-F-I-H L-A-F-I-H P-H-S-E L-A-F-I-H L-A-F-I-H
        L-A-F-I-H L-A-F-I-H L-A-F-I-H L-A-F-I-H L-A-F-I-H L-A-F-I-H
        L-A-F-I-H L-A-F-I-H L-A-F-I-H L-A-F-I-H L-A-F-I-H L-A-F-I-H
        R-C-O F-C-O-P C-E D-A-P F-C-P R-C-P ]
       (83 163 167 170)   (171 176))
(P-O-I-P
       [P-P-D-I-H S-P O-I-H-I-V L-A-F-I-P L-A-F-I-H L-A-F-I-H
        L-A-F-I-H L-A-F-I-H L-A-F-I-H L-A-F-I-H L-A-F-I-H L-A-F-I-H
        P-H-S-E L-A-F-I-H L-A-F-I-H L-A-F-I-H L-A-F-I-H L-A-F-I-H
        L-A-F-I-H L-A-F-I-H L-A-F-I-H L-A-F-I-H L-A-F-I-H L-A-F-I-H
        L-A-F-I-H L-A-F-I-H L-A-F-I-H R-C-O F-C-O-P C-E
        D-A-P F-C-P R-C-P ]
       (174 175 176)   (177 181))
                           .
                           .
                           .
(THIS SEQUENCE REPEATED 22 TIMES)
                           .
                           .
                           .
(E-H
       [L-A-F-I-P R-C-O F-C-O-P C-E D-A-P F-C-P
        R-C-P ]
       (152 465 467)   (468 468))
>>>>>
```

- 123 -

BUILD A PILE OF PIECES WITH STRAIGHT EDGES AND START PUZZLE
SET UP THE PILE FIRST:

<<<<<
  (R-C-P
    [L-A-F-I-P R-C-O F-C-O-P C-E D-A-P F-C-P]
    (152 465 467) (469 470))
  (L-A-F-I-P
    [R-P-I-C R-C-P F-R-P R-C-O F-C-O-P C-E
    D-A-P F-C-P]
    (152 470) (471 472))
  (R-C-O
    [F-O-F-P L-A-N-I-P F-H-S-E F-H-S-E F-C-O-P C-E
    R-P-I-C R-C-P F-R-P D-A-P F-C-P]
    (152 153 465 466 472) (473 474))
  (L-A-N-I-P
    [R-O-I-V R-C-O F-R-O P-O-F-P F-H-S-E F-H-S-E
    F-C-O-P C-E R-P-I-C R-C-P F-R-P D-A-P
    F-C-P]
    (472 474) (475 478))
  (P-O-F-P
    [L-A-N-I-P R-C-O F-H-S-E F-C-O-P C-E D-A-P
    F-R-O R-P-I-C R-C-P F-R-P F-C-P]
    (152 153 466 477 478) (479 483))
  (F-C-P
    [P-O-I-P S-P O-I-H-I-V L-A-F-I-P R-C-O F-H-S-E
    F-C-O-P C-E D-A-P F-R-O R-P-I-C R-C-P
    F-R-P]
    (152 153 481 482 483) (484 487))
  (M-A-P
    [F-A-P S-P F-H-S-E O-I-H-I-V R-C-O F-C-O-P
    C-E F-R-O F-R-P]
    (155 482 483 486 487) (488 497))
  (F-C-P
    [P-O-F-P R-C-O F-H-S-E F-C-O-P C-E D-A-P
    R-C-P F-R-O F-R-P]
    (492 493 494 495 496) (498 501))
  (F-A-P
    [R-C-O F-H-S-E F-C-O-P C-E F-R-O F-R-P]
    (500 501) (502 504))
  (L-A-F-I-P
    [D-A-P F-C-P R-P-I-C R-C-P R-C-O F-H-S-E
    F-C-O-P C-E F-R-O F-R-P]
>>>>>   (481 503 504) (505 506))

- 124 -

NOW BUILD THE FILE:

```
<<<<<
  (L-A-N-I-F
     [D-A-F F-C-F R-F-I-C R-C-F R-C-O F-C-O-F
      C-E F-R-O F-R-F ]
     (503 504 506)  (507 508))
  (F-O-F-F
     [L-A-N-I-F R-C-O F-H-S-E F-C-O-F C-E D-A-F
      F-C-F R-F-I-C R-C-F F-R-O F-R-F ]
     (481 492 503 504 506)  (509 511))
  (F-C-F
     [F-O-I-F S-F O-I-H-I-V R-C-O F-H-S-E F-C-O-F
      C-E D-A-F R-F-I-C R-C-F F-R-O F-R-F ]
     (503 504 509 510 511)  (512 515))
  (F-A-F
     [M-A-F S-F O-I-H-I-V R-C-O F-H-S-E F-C-O-F
      C-E F-R-O F-R-F ]
     (497 510 511 514 515)  (516 518))
  (F-O-I-F
     [D-A-F F-C-F L-A-F-I-F R-C-F S-F O-I-H-I-V
      R-C-O F-H-S-E F-C-O-F C-E F-R-O F-R-F ]
     (493 510 511 517 518)  (519 523))
  (F-C-F
     [L-A-F-I-F R-C-O F-H-S-E F-C-O-F C-E D-A-F
      R-C-F F-R-O F-R-F ]
     (517 521 523)  (524 527))
  (F-A-F
     [R-C-O F-H-S-E F-C-O-F C-E F-R-O F-R-F ]
     (526 527)  (528 530))
  (L-A-F-I-F
     [D-A-F F-C-F R-F-I-C R-C-F R-C-O F-H-S-E
      F-C-O-F C-E F-R-O F-R-F ]
     (509 529 530)  (531 532))
                         .
                         .
                         .
  (THIS SEQUENCE REPEATED 13 TIMES)
                         .
                         .
                         .
  (F-O-F-F
     [L-A-N-I-F R-O-I-V R-C-O F-H-S-E F-H-S-E F-C-O-F
      C-E D-A-F F-C-F R-F-I-C R-C-F F-R-O
      F-R-F ]
     (892 899 900 905 906)  (907 911))
  (F-C-F
     [F-O-I-F S-F O-I-H-I-V L-A-F-I-F R-O-I-V R-C-O
      F-H-S-E F-H-S-E F-C-O-F C-E R-O-I-H D-A-F
      R-F-I-C R-C-F F-R-O F-R-F ]
     (899 900 909 910 911)  (912 915))
  (F-A-F
     [M-A-F S-F O-I-H-I-V R-O-I-V R-C-O F-H-S-E
      F-H-S-E F-C-O-F C-E R-O-I-H F-R-O F-R-F ]
     (497 910 911 914 915)  (916 918))
  (F-O-I-F
     [D-A-F F-C-F L-A-F-I-F R-C-F S-F O-I-H-I-V
      R-O-I-V R-C-O F-H-S-E F-H-S-E F-C-O-F C-E
      R-O-I-H F-R-O F-R-F ]
```

```
      (891 910 911 917 918)   (919 923))
(F-R-O
   [L-A-F-I-P R-O-I-V R-C-O P-H-S-E P-H-S-E F-C-O-P
    C-E D-A-P F-C-P R-C-P F-R-P ]
   (917 921 923)   (924 925))
(L-A-F-I-P
   [R-C-O P-H-S-E P-H-S-E F-C-O-P C-E D-A-P
    F-C-P R-C-P F-R-P ]
   (923 925)   (926 929))
(F-O-F-P
   [R-O-I-V R-C-O L-A-N-I-P P-H-S-E F-C-O-P C-E
    D-A-P F-C-P R-C-P F-R-P ]
   (927 929)   (930 934))
(S-P
   [P-O-I-P O-I-H-I-V L-A-F-I-P R-C-O P-H-S-E F-C-O-P
    C-E D-A-P F-C-P R-C-P F-R-P ]
   (917 918 932 933 934)   (935 941))
(L-A-F-I-P
   [D-A-P P-H-S-E F-C-P R-C-O R-C-P F-R-P
    F-C-O-P C-E ]
   (8 11 936)   (942 943))
>.>>>
```

BUILD OUTSIDE EDGES OF PUZZLE FIRST:

<<<<<
  (L-A-N-I-P
    [P-O-F-P P-H-S-E R-C-O F-C-O-P C-E D-A-P
     F-C-P R-C-P F-R-P]
    (6 8 11 113 943) (944 947))
  (P-O-F-P
    [L-A-N-I-P P-F-I-P P-H-S-E R-C-O F-C-O-P C-E
     D-A-P F-C-P R-C-P F-R-P]
    (6 8 11 946 947) (948 952))
  (F-P-I-P
    [P-O-I-P O-I-H-I-V L-A-F-I-P P-F-I-P P-H-S-E R-C-O
     F-C-O-P C-E D-A-P F-C-P R-C-P F-R-P]
    (8 11 950 951 952) (953 957))
  (P-F-I-P
    [F-F-I-P P-O-I-P O-I-H-I-V L-A-F-I-P P-F-I-P P-H-S-E
     R-C-O F-C-O-P C-E D-A-P F-C-P R-C-P
     F-R-P]
    (114 951 952 957) (958 965))
  (L-A-F-I-P
    [D-A-P F-C-P R-C-P F-R-P]
    (8 950 964) (966 967))
                       .
                       .
                       .
  (THIS SEQUENCE REPEATED 14 TIMES)
                       .
                       .
                       .
  (P-O-F-P
    [P-F-I-P P-H-S-E P-H-S-E R-C-O F-C-O-P C-E
     D-A-P F-C-P R-C-P F-R-P]
    (8 11 1282 1297 1299) (1300 1304))
  (F-P-I-P
    [F-O-I-P F-F-I-P O-I-H-I-V P-H-S-E P-H-S-E R-C-O
     F-C-O-P C-E D-A-P P-I-E F-C-P R-C-P
     F-R-P]
    (8 11 1302 1303 1304) (1305 1311))
  (F-P-I-P
    [F-F-I-P F-O-I-P P-F-I-P O-I-H-I-V P-H-S-E P-H-S-E
     R-C-O F-C-O-P C-E D-A-P P-I-E F-C-P
     R-C-P F-R-P]
    (1294 1303 1304 1311) (1312 1319))
>>>>>

```
SET UP A FILE:

<<<<<
  (F-C-F
    [D-A-F F-I-E R-C-F F-R-F ]
    (8 11 1302)  (1320 1323))
  (F-A-F
    [F-R-F ]
    (1322 1323)   (1324 1326))
  (L-A-F-I-F
    [D-A-F F-C-F R-F-I-C R-C-F F-R-F ]
    (112 1325 1326)   (1327 1328))
  (R-C-O
    [F-O-F-F L-A-N-I-F F-C-O-F C-E D-A-F F-C-F
     R-F-I-C R-C-F F-R-F ]
    (112 1319 1325 1326 1328)   (1329 1330))
  (F-C-O-F
    [R-O-I-V R-C-O F-R-O F-O-F-F L-A-N-I-F C-E
     D-A-F F-C-F R-F-I-C R-C-F F-R-F ]
    (1328 1330)   (1331 1332))
  (L-A-N-I-F
    [F-H-C-C FT-C-O-F F-C-O-F R-O-I-V R-C-O F-R-O
     F-O-F-F C-E D-A-F F-C-F R-F-I-C R-C-F
     F-R-F ]
    (1328 1332)   (1333 1335))
>>>>>
```

BUILD A FILE OF PIECES WITH THE SAME COLOR:

<<<<<
  (L-A-N-I-P
    [P-H-C-C FT-C-O-P F-C-O-P R-C-O F-R-O P-O-F-P
     C-E D-A-P F-C-P R-P-I-C R-C-P F-R-P]
    (1334 1335) (1336 1337))
  (P-O-F-P
    [L-A-N-I-P P-F-I-P P-H-C-C FT-C-O-P F-C-O-P R-C-O
     C-E D-A-P F-R-O F-C-P R-P-I-C R-C-P
     F-R-P]
    (1319 1325 1326 1336 1337) (1338 1341))
  (F-C-P
    [P-O-I-P P-F-I-P O-I-H-I-V L-A-F-I-P P-F-I-P P-H-C-C
     FT-C-O-P F-C-O-P R-C-O C-E D-A-P F-R-O
     R-P-I-C R-C-P F-R-P]
    (1325 1326 1339 1340 1341) (1342 1345))
  (P-A-P
    [M-A-P F-F-I-P O-I-H-I-V P-F-I-P P-H-C-C FT-C-O-P
     F-C-O-P R-C-O C-E F-R-O F-R-P]
    (3 1340 1341 1344 1345) (1346 1348))
  (P-O-I-P
    [D-A-P P-I-E F-C-P R-C-P F-F-I-P O-I-H-I-V
     P-F-I-P P-H-C-C FT-C-O-P F-C-O-P R-C-O C-E
     F-R-O F-R-P]
    (1302 1340 1341 1347 1348) (1349 1353))
  (F-C-P
    [P-O-F-P P-F-I-P P-H-C-C FT-C-O-P F-C-O-P R-C-O
     C-E D-A-P R-C-P F-R-O F-R-P]
    (1347 1348 1351 1352 1353) (1354 1357))
  (P-A-P
    [P-F-I-P P-H-C-C FT-C-O-P F-C-O-P R-C-O C-E
     F-R-O F-R-P]
    (1356 1357) (1358 1360))
  (L-A-F-I-P
    [D-A-P F-C-P R-P-I-C R-C-P P-F-I-P P-H-C-C
     FT-C-O-P F-C-O-P R-C-O C-E F-R-O F-R-P]
    (1339 1359 1360) (1361 1362))
                      .
                      .
                      .
(THIS SEQUENCE REPEATED 5 TIMES)
                      .
                      .
                      .
  (F-R-O
    [P-O-F-P L-A-N-I-P P-H-C-C FT-C-O-P F-C-O-P R-O-I-V
     R-C-O C-E D-A-P F-C-P R-P-I-C R-C-P
     F-R-P]
    (1473 1486 1493 1494 1496) (1497 1498))
  (P-O-F-P
    [R-C-O L-A-N-I-P P-H-C-C FT-C-O-P F-C-O-P C-E
     D-A-P F-C-P R-P-I-C R-C-P F-R-P]
    (1496 1498) (1499 1503))
  (F-C-P
    [P-O-I-P O-I-H-I-V L-A-F-I-P R-C-O P-H-C-C FT-C-O-P
     F-C-O-P C-E D-A-P R-P-I-C R-C-P F-R-P]
    (1493 1494 1501 1502 1503) (1504 1507))
  (P-A-P

- 129 -

```
[M-A-P O-I-H-I-V R-C-O P-H-C-C FT-C-O-P F-C-O-P
 C-E F-R-P ]
  (3 1502 1503 1506 1507)  (1508 1510))
(P-O-I-P
 [D-A-P F-C-P L-A-F-I-P R-C-P O-I-H-I-V R-C-O
  P-H-C-C FT-C-O-P F-C-O-P C-E F-R-P ]
  (1485 1502 1503 1509 1510)  (1511 1515))
>>>>>
```

BUILD PUZZLE FROM PIECES WITH COMMON COLOR:

<<<<<
  (L-A-F-I-P
    [D-A-P F-C-P R-C-P F-R-P]
    (1509 1513 1515) (1516 1517))
  (P-O-F-P
    [L-A-N-I-P P-F-I-P R-C-O P-H-C-C FT-C-O-P F-C-O-P
     C-E D-A-P F-C-P R-C-P F-R-P]
    (1509 1510 1513 1514 1517) (1518 1522))
  (F-P-I-P
    [P-O-I-P O-I-H-I-V L-A-F-I-P P-F-I-P R-C-O P-H-C-C
     FT-C-O-P F-C-O-P C-E D-A-P F-C-P R-C-P
     F-R-P]
    (1509 1510 1520 1521 1522) (1523 1531))
  (P-P-I-P
    [F-P-I-P P-O-I-P O-I-H-I-V L-A-F-I-P P-F-I-P R-C-O
     P-H-C-C FT-C-O-P F-C-O-P C-E D-A-P F-C-P
     R-C-P F-R-P]
    (1509 1521 1522 1531) (1532 1539))
                         .
                         .
                         .
(THIS SEQUENCE REPEATED 5 TIMES)
                         .
                         .
                         .
  (L-A-F-I-P
    [D-A-P F-C-P R-C-P F-R-P]
    (1509 1634 1654) (1656 1657))
  (P-O-F-P
    [F-F-I-P R-C-O P-H-C-C FT-C-O-P F-C-O-P C-E
     D-A-P F-C-P R-C-P F-R-P]
    (1509 1510 1634 1655 1657) (1658 1662))
  (F-P-I-P
    [P-O-I-P P-F-I-P O-I-H-I-V R-C-O P-H-C-C FT-C-O-P
     F-C-O-P C-E D-A-P P-I-E F-C-P R-C-P
     F-R-P]
    (1509 1510 1660 1661 1662) (1663 1671))
  (F-P-I-P
    [F-F-I-P P-O-I-P P-F-I-P O-I-H-I-V R-C-O P-H-C-C
     FT-C-O-P F-C-O-P C-E D-A-P P-I-E F-C-P
     R-C-P F-R-P]
    (1652 1661 1662 1671) (1672 1679))
>>>>>

WHAT FOLLOWED WAS A REPETITION OF THE PRECEEDING SEQUENCES:
        SET UP A FILE
        BUILD A FILE OF PIECES WITH SAME COLOR
        BUILD THE PUZZLE FROM THIS FILE
            .
            .
            .

    FINALLY, CLEAN UP AND STOP:

<<<<<
    (P-C-P
        [P-I-P D-A-P P-I-E R-C-P F-R-P]
        (8 11 1732) (1786 1789))
    (P-A-P
        [P-I-P F-R-P]
        (1732 1787) (1790 1792))
    (D-A-P
        [P-I-P F-C-P P-I-E R-C-P F-R-P]
        (1787 1791 1792) (1793 1797))
    (P-A-P
        [P-I-P F-R-P]
        (3 1794 1796) (1798 1800))
    (D-A-P
        [P-I-P P-I-E F-C-P R-C-P F-R-P]
        (2 1799 1800) (1801 1805))
    (F-R-P
        [T-A-N-P P-I-P]
        (470 1804 1805) (1806 1807))
    (P-I-P
        [T-A-N-P]
        (1804 1805) (1808 1808))
>>>>>

            *** END OF TRACE ***

APPENDIX 2
==========

PRETTY PRINTED CRAPS OUTPUT FOR PUZZLE PS.

```
<<
  (LOOK-AT-PIECE-IN-HEAP
      (THERE-ARE-NO-PILES))
  (PICK-UP-OBJECT-IN-VIEW
      (LOOK-AT-PIECE-IN-HEAP
       CLOSE-EYES
       FIND-COLOR-OF-PIECE
       THERE-ARE-NO-PILES
       REMEMBER-CURRENT-OBJECT))
  (MAKE-A-PILE
      (LOOK-AT-PIECE-IN-HEAP
       CLOSE-EYES
       OBJECT-IN-HAND-IN-VIEW
       PUT-PIECE-DOWN-IN-HEAP
       FIND-COLOR-OF-PIECE
       START-PUZZLE
       THERE-ARE-NO-PILES
       REMEMBER-CURRENT-OBJECT))
  [REPEAT    (WHILE   (LOOK-AT-PIECE-IN-HEAP))
             (UNTIL   (EMPTY-HEAP))
       <<
       (LOOK-AT-PIECE-IN-HEAP
           (OR.*
            (CLOSE-EYES
             FIND-COLOR-OF-PIECE
             PIECE-HAS-STRAIGHT-EDGE
             FORGET-CURRENT-PILE
             DESTROY-A-PILE
             REMEMBER-CURRENT-PILE
             REMEMBER-CURRENT-OBJECT)
            (CLOSE-EYES
             FIND-COLOR-OF-PIECE
             FORGET-CURRENT-PILE
             DESTROY-A-PILE
             REMEMBER-CURRENT-PILE
             REMEMBER-CURRENT-OBJECT)
            (CLOSE-EYES
             FIND-COLOR-OF-PIECE
             PIECE-HAS-STRAIGHT-EDGE
             PICK-OBJECT-FROM-PILE
             FORGET-CURRENT-PILE
             DESTROY-A-PILE
             REMEMBER-CURRENT-PILE
             REMEMBER-CURRENT-OBJECT)))
       (PICK-UP-OBJECT-IN-VIEW
           (OR.*
            (CLOSE-EYES
             FIND-COLOR-OF-PIECE
             FORGET-CURRENT-PILE
             DESTROY-A-PILE
             LOOK-AT-FIRST-IN-PILE
             REMEMBER-CURRENT-PILE
             REMEMBER-CURRENT-OBJECT)
```

```
                    (LOOK-AT-PIECE-IN-HEAP
                     CLOSE-EYES
                     FIND-COLOR-OF-PIECE
                     PIECE-HAS-STRAIGHT-EDGE
                     FORGET-CURRENT-PILE
                     DESTROY-A-PILE
                     LOOK-AT-FIRST-IN-FILE
                     REMEMBER-CURRENT-FILE
                     REMEMBER-CURRENT-OBJECT)
                    (LOOK-AT-PIECE-IN-HEAP
                     CLOSE-EYES
                     FIND-COLOR-OF-PIECE
                     FORGET-CURRENT-PILE
                     DESTROY-A-PILE
                     LOOK-AT-FIRST-IN-FILE
                     REMEMBER-CURRENT-FILE
                     REMEMBER-CURRENT-OBJECT)))
                (PUT-OBJECT-IN-FILE
                    (OR.*
                    (CLOSE-EYES
                     OBJECT-IN-HAND-IN-VIEW
                     PUT-PIECE-DOWN-IN-HEAP
                     EMPTY-HEAP
                     FIND-COLOR-OF-PIECE
                     START-PUZZLE
                     FORGET-CURRENT-FILE
                     DESTROY-A-PILE
                     LOOK-AT-FIRST-IN-FILE
                     REMEMBER-CURRENT-FILE
                     REMEMBER-CURRENT-OBJECT)
                    (LOOK-AT-PIECE-IN-HEAP
                     CLOSE-EYES
                     OBJECT-IN-HAND-IN-VIEW
                     PUT-PIECE-DOWN-IN-HEAP
                     FIND-COLOR-OF-PIECE
                     PIECE-HAS-STRAIGHT-EDGE
                     START-PUZZLE
                     FORGET-CURRENT-FILE
                     DESTROY-A-PILE
                     LOOK-AT-FIRST-IN-FILE
                     REMEMBER-CURRENT-FILE
                     REMEMBER-CURRENT-OBJECT)
                    (LOOK-AT-PIECE-IN-HEAP
                     CLOSE-EYES
                     OBJECT-IN-HAND-IN-VIEW
                     PUT-PIECE-DOWN-IN-HEAP
                     FIND-COLOR-OF-PIECE
                     START-PUZZLE
                     FORGET-CURRENT-FILE
                     DESTROY-A-PILE
                     LOOK-AT-FIRST-IN-FILE
                     REMEMBER-CURRENT-FILE
                     REMEMBER-CURRENT-OBJECT)))
                >>
            )
            (EMPTY-HEAP
                (CLOSE-EYES
                 FIND-COLOR-OF-PIECE
                 FORGET-CURRENT-PILE
                 DESTROY-A-PILE
```

```
                    LOOK-AT-FIRST-IN-FILE
                    REMEMBER-CURRENT-FILE
                    REMEMBER-CURRENT-OBJECT))
            (REMEMBER-CURRENT-FILE
                    (CLOSE-EYES
                    FIND-COLOR-OF-PIECE
                    FORGET-CURRENT-FILE
                    DESTROY-A-FILE
                    LOOK-AT-FIRST-IN-FILE
                    REMEMBER-CURRENT-OBJECT))
            (LOOK-AT-FIRST-IN-FILE
                    (CLOSE-EYES
                    FIND-COLOR-OF-PIECE
                    FORGET-CURRENT-FILE
                    DESTROY-A-FILE
                    REMEMBER-CURRENT-FILE
                    REMEMBERED-FILE-IS-CURRENT
                    FORGET-REMEMBERED-FILE
                    REMEMBER-CURRENT-OBJECT))
            (REMEMBER-CURRENT-OBJECT
                    (CLOSE-EYES
                    FIND-COLOR-OF-PIECE
                    PIECE-HAS-STRAIGHT-EDGE
                    PICK-OBJECT-FROM-FILE
                    FORGET-CURRENT-FILE
                    DESTROY-A-FILE
                    LOOK-AT-NEXT-IN-FILE
                    REMEMBER-CURRENT-FILE
                    REMEMBERED-FILE-IS-CURRENT
                    FORGET-REMEMBERED-FILE))
            (LOOK-AT-NEXT-IN-FILE
                    (CLOSE-EYES
                    FIND-COLOR-OF-PIECE
                    PIECE-HAS-STRAIGHT-EDGE
                    PICK-OBJECT-FROM-FILE
                    FORGET-CURRENT-FILE
                    DESTROY-A-FILE
                    REMEMBER-CURRENT-FILE
                    REMEMBERED-FILE-IS-CURRENT
                    FORGET-REMEMBERED-FILE
                    REMEMBER-CURRENT-OBJECT
                    FORGET-REMEMBERED-OBJECT
                    REMEMBERED-OBJECT-IN-VIEW))
            (PICK-OBJECT-FROM-FILE
                    (CLOSE-EYES
                    FIND-COLOR-OF-PIECE
                    PIECE-HAS-STRAIGHT-EDGE
                    FORGET-CURRENT-FILE
                    DESTROY-A-FILE
                    LOOK-AT-NEXT-IN-FILE
                    REMEMBER-CURRENT-FILE
                    REMEMBERED-FILE-IS-CURRENT
                    FORGET-REMEMBERED-FILE
                    REMEMBER-CURRENT-OBJECT
                    FORGET-REMEMBERED-OBJECT))
            (FORGET-CURRENT-FILE
                    (CLOSE-EYES
                    OBJECT-IN-HAND-IN-VIEW
                    FIND-COLOR-OF-PIECE
                    PIECE-HAS-STRAIGHT-EDGE
```

```
                        START-PUZZLE
                        PUT-OBJECT-IN-PILE
                        DESTROY-A-PILE
                        LOOK-AT-FIRST-IN-PILE
                        REMEMBER-CURRENT-PILE
                        REMEMBERED-PILE-IS-CURRENT
                        FORGET-REMEMBERED-PILE
                        REMEMBER-CURRENT-OBJECT
                        FORGET-REMEMBERED-OBJECT))
                  (MAKE-A-PILE
                        (CLOSE-EYES
                        OBJECT-IN-HAND-IN-VIEW
                        FIND-COLOR-OF-PIECE
                        START-PUZZLE
                        PICK-A-FILE
                        FORGET-REMEMBERED-PILE
                        REMEMBER-CURRENT-OBJECT
                        FORGET-REMEMBERED-OBJECT))
                  (FORGET-CURRENT-FILE
                        (CLOSE-EYES
                        FIND-COLOR-OF-PIECE
                        PIECE-HAS-STRAIGHT-EDGE
                        PICK-OBJECT-FROM-FILE
                        DESTROY-A-PILE
                        REMEMBER-CURRENT-PILE
                        FORGET-REMEMBERED-PILE
                        REMEMBER-CURRENT-OBJECT
                        FORGET-REMEMBERED-OBJECT))
                  (PICK-A-FILE
                        (CLOSE-EYES
                        FIND-COLOR-OF-PIECE
                        PIECE-HAS-STRAIGHT-EDGE
                        FORGET-REMEMBERED-FILE
                        REMEMBER-CURRENT-OBJECT
                        FORGET-REMEMBERED-OBJECT))
                  (LOOK-AT-FIRST-IN-FILE
                        (CLOSE-EYES
                        FIND-COLOR-OF-PIECE
                        PIECE-HAS-STRAIGHT-EDGE
                        FORGET-CURRENT-FILE
                        DESTROY-A-PILE
                        REMEMBER-CURRENT-PILE
                        REMEMBERED-FILE-IS-CURRENT
                        FORGET-REMEMBERED-FILE
                        REMEMBER-CURRENT-OBJECT
                        FORGET-REMEMBERED-OBJECT))
            [REPEAT    (WHILE   NIL)
                        (UNTIL   (PIECE-HAS-STRAIGHT-EDGE
                                    REMEMBERED-OBJECT-IN-VIEW))
                  <<
                  [REPEAT     (WHILE   NIL)
                              (UNTIL   (PIECE-HAS-STRAIGHT-EDGE))
                        <<
                        (LOOK-AT-NEXT-IN-FILE
                              (CLOSE-EYES
                              FIND-COLOR-OF-PIECE
                              PICK-OBJECT-FROM-FILE
                              FORGET-CURRENT-FILE
                              DESTROY-A-PILE
                              REMEMBER-CURRENT-PILE
```

```
                     REMEMBERED-FILE-IS-CURRENT
                     FORGET-REMEMBERED-FILE
                     REMEMBER-CURRENT-OBJECT
                     FORGET-REMEMBERED-OBJECT))
           >>
     ]
     [REPEAT    (WHILE  (PIECE-HAS-STRAIGHT-EDGE))
                (UNTIL  (REMEMBERED-OBJECT-IN-VIEW))
           <<
         (PICK-OBJECT-FROM-FILE
              (CLOSE-EYES
               FIND-COLOR-OF-PIECE
               PIECE-HAS-STRAIGHT-EDGE
               FORGET-CURRENT-FILE
               DESTROY-A-FILE
               LOOK-AT-NEXT-IN-FILE
               REMEMBER-CURRENT-FILE
               REMEMBERED-FILE-IS-CURRENT
               FORGET-REMEMBERED-FILE
               REMEMBER-CURRENT-OBJECT
               FORGET-REMEMBERED-OBJECT))
         (FORGET-CURRENT-FILE
              (CLOSE-EYES
               OBJECT-IN-HAND-IN-VIEW
               FIND-COLOR-OF-PIECE
               PIECE-HAS-STRAIGHT-EDGE
               START-PUZZLE
               PUT-OBJECT-IN-FILE
               DESTROY-A-FILE
               LOOK-AT-FIRST-IN-FILE
               REMEMBER-CURRENT-FILE
               REMEMBERED-FILE-IS-CURRENT
               FORGET-REMEMBERED-FILE
               REMEMBER-CURRENT-OBJECT
               FORGET-REMEMBERED-OBJECT))
         (PICK-A-FILE
              (CLOSE-EYES
               OBJECT-IN-HAND-IN-VIEW
               FIND-COLOR-OF-PIECE
               PIECE-HAS-STRAIGHT-EDGE
               START-PUZZLE
               MAKE-A-FILE
               FORGET-REMEMBERED-FILE
               REMEMBER-CURRENT-OBJECT
               FORGET-REMEMBERED-OBJECT))
         (PUT-OBJECT-IN-FILE
              (CLOSE-EYES
               OBJECT-IN-HAND-IN-VIEW
               FIND-COLOR-OF-PIECE
               PIECE-HAS-STRAIGHT-EDGE
               START-PUZZLE
               FORGET-CURRENT-FILE
               DESTROY-A-FILE
               LOOK-AT-FIRST-IN-FILE
               REMEMBER-CURRENT-FILE
               FORGET-REMEMBERED-FILE
               REMEMBER-CURRENT-OBJECT
               FORGET-REMEMBERED-OBJECT))
         (FORGET-CURRENT-FILE
              (CLOSE-EYES
```

```
                              FIND-COLOR-OF-PIECE
                              PIECE-HAS-STRAIGHT-EDGE
                              DESTROY-A-FILE
                              LOOK-AT-FIRST-IN-PILE
                              REMEMBER-CURRENT-PILE
                              FORGET-REMEMBERED-FILE
                              REMEMBER-CURRENT-OBJECT
                              FORGET-REMEMBERED-OBJECT))
                       (PICK-A-FILE
                              (CLOSE-EYES
                              FIND-COLOR-OF-PIECE
                              PIECE-HAS-STRAIGHT-EDGE
                              FORGET-REMEMBERED-FILE
                              REMEMBER-CURRENT-OBJECT
                              FORGET-REMEMBERED-OBJECT))
                       (LOOK-AT-FIRST-IN-FILE
                              (CLOSE-EYES
                              FIND-COLOR-OF-PIECE
                              PIECE-HAS-STRAIGHT-EDGE
                              FORGET-CURRENT-FILE
                              DESTROY-A-FILE
                              REMEMBER-CURRENT-FILE
                              REMEMBERED-FILE-IS-CURRENT
                              FORGET-REMEMBERED-FILE
                              REMEMBER-CURRENT-OBJECT
                              FORGET-REMEMBERED-OBJECT))
                 >>
            ]
            >>
      ]
      (PICK-OBJECT-FROM-FILE
            (CLOSE-EYES
            FIND-COLOR-OF-PIECE
            PIECE-HAS-STRAIGHT-EDGE
            FORGET-CURRENT-FILE
            DESTROY-A-FILE
            LOOK-AT-NEXT-IN-FILE
            REMEMBER-CURRENT-FILE
            REMEMBERED-FILE-IS-CURRENT
            FORGET-REMEMBERED-FILE
            REMEMBER-CURRENT-OBJECT
            FORGET-REMEMBERED-OBJECT
            REMEMBERED-OBJECT-IN-VIEW))
      (FORGET-CURRENT-FILE
            (CLOSE-EYES
            OBJECT-IN-HAND-IN-VIEW
            FIND-COLOR-OF-PIECE
            PIECE-HAS-STRAIGHT-EDGE
            START-PUZZLE
            PUT-OBJECT-IN-PILE
            DESTROY-A-FILE
            LOOK-AT-FIRST-IN-FILE
            REMEMBER-CURRENT-FILE
            REMEMBERED-FILE-IS-CURRENT
            FORGET-REMEMBERED-FILE
            REMEMBER-CURRENT-OBJECT
            FORGET-REMEMBERED-OBJECT
            REMEMBERED-OBJECT-IN-VIEW
            REMEMBERED-OBJECT-IN-HAND))
      (PICK-A-FILE
```

- 138 -

```
            (CLOSE-EYES
             OBJECT-IN-HAND-IN-VIEW
             FIND-COLOR-OF-PIECE
             PIECE-HAS-STRAIGHT-EDGE
             START-PUZZLE
             MAKE-A-FILE
             FORGET-REMEMBERED-FILE
             REMEMBER-CURRENT-OBJECT
             FORGET-REMEMBERED-OBJECT
             REMEMBERED-OBJECT-IN-VIEW
             REMEMBERED-OBJECT-IN-HAND))
        (PUT-OBJECT-IN-FILE
            (CLOSE-EYES
             OBJECT-IN-HAND-IN-VIEW
             FIND-COLOR-OF-PIECE
             PIECE-HAS-STRAIGHT-EDGE
             START-PUZZLE
             FORGET-CURRENT-FILE
             DESTROY-A-FILE
             LOOK-AT-FIRST-IN-FILE
             REMEMBER-CURRENT-FILE
             FORGET-REMEMBERED-FILE
             REMEMBER-CURRENT-OBJECT
             FORGET-REMEMBERED-OBJECT
             REMEMBERED-OBJECT-IN-VIEW
             REMEMBERED-OBJECT-IN-HAND))
        (FORGET-REMEMBERED-OBJECT
            (CLOSE-EYES
             FIND-COLOR-OF-PIECE
             PIECE-HAS-STRAIGHT-EDGE
             FORGET-CURRENT-FILE
             DESTROY-A-FILE
             LOOK-AT-FIRST-IN-FILE
             REMEMBER-CURRENT-FILE
             FORGET-REMEMBERED-FILE
             REMEMBER-CURRENT-OBJECT
             REMEMBERED-OBJECT-IN-VIEW))
        (LOOK-AT-FIRST-IN-FILE
            (CLOSE-EYES
             FIND-COLOR-OF-PIECE
             PIECE-HAS-STRAIGHT-EDGE
             FORGET-CURRENT-FILE
             DESTROY-A-FILE
             REMEMBER-CURRENT-FILE
             FORGET-REMEMBERED-FILE
             REMEMBER-CURRENT-OBJECT))
        (PICK-OBJECT-FROM-FILE
            (CLOSE-EYES
             FIND-COLOR-OF-PIECE
             PIECE-HAS-STRAIGHT-EDGE
             FORGET-CURRENT-FILE
             DESTROY-A-FILE
             LOOK-AT-NEXT-IN-FILE
             REMEMBER-CURRENT-FILE
             FORGET-REMEMBERED-FILE
             REMEMBER-CURRENT-OBJECT))
        (START-PUZZLE
            (CLOSE-EYES
             OBJECT-IN-HAND-IN-VIEW
             FIND-COLOR-OF-PIECE
```

```
        PIECE-HAS-STRAIGHT-EDGE
        PUT-OBJECT-IN-FILE
        FORGET-CURRENT-FILE
        DESTROY-A-FILE
        LOOK-AT-FIRST-IN-FILE
        REMEMBER-CURRENT-FILE
        FORGET-REMEMBERED-FILE
        REMEMBER-CURRENT-OBJECT))
 (LOOK-AT-FIRST-IN-FILE
        (FORGET-CURRENT-FILE
        DESTROY-A-FILE
        REMEMBER-CURRENT-FILE
        FORGET-REMEMBERED-FILE))
 EREPEAT    (WHILE  (LOOK-AT-NEXT-IN-FILE))
            (UNTIL  NIL)
     <<
     EREPEAT    (WHILE  NIL)
                (UNTIL  (PIECE-FITS-IN-PUZZLE))
        <<
        (LOOK-AT-NEXT-IN-FILE
            (CLOSE-EYES
            FIND-COLOR-OF-PIECE
            PIECE-HAS-STRAIGHT-EDGE
            PICK-OBJECT-FROM-FILE
            FORGET-CURRENT-FILE
            DESTROY-A-FILE
            REMEMBER-CURRENT-FILE
            FORGET-REMEMBERED-FILE
            REMEMBER-CURRENT-OBJECT))
        >>
     ]
     EREPEAT    (WHILE  (LOOK-AT-NEXT-IN-FILE
                        PIECE-FITS-IN-PUZZLE))
                (UNTIL  NIL)
        <<
        (PICK-OBJECT-FROM-FILE
            (CLOSE-EYES
            FIND-COLOR-OF-PIECE
            PIECE-HAS-STRAIGHT-EDGE
            PIECE-FITS-IN-PUZZLE
            FORGET-CURRENT-FILE
            DESTROY-A-FILE
            LOOK-AT-NEXT-IN-FILE
            REMEMBER-CURRENT-FILE
            FORGET-REMEMBERED-FILE
            REMEMBER-CURRENT-OBJECT))
        (FIT-PIECE-IN-PUZZLE
            (CLOSE-EYES
            OBJECT-IN-HAND-IN-VIEW
            FIND-COLOR-OF-PIECE
            PIECE-HAS-STRAIGHT-EDGE
            PIECE-FITS-IN-PUZZLE
            PUT-OBJECT-IN-FILE
            FORGET-CURRENT-FILE
            DESTROY-A-FILE
            LOOK-AT-FIRST-IN-FILE
            REMEMBER-CURRENT-FILE
            FORGET-REMEMBERED-FILE
            REMEMBER-CURRENT-OBJECT))
        (PIECE-PUT-IN-PUZZLE
```

```
                      (CLOSE-EYES
                       OBJECT-IN-HAND-IN-VIEW
                       FIND-COLOR-OF-PIECE
                       PIECE-HAS-STRAIGHT-EDGE
                       PIECE-FITS-IN-PUZZLE
                       FIT-PIECE-IN-PUZZLE
                       PUT-OBJECT-IN-FILE
                       FORGET-CURRENT-FILE
                       DESTROY-A-FILE
                       LOOK-AT-FIRST-IN-FILE
                       REMEMBER-CURRENT-FILE
                       FORGET-REMEMBERED-FILE
                       REMEMBER-CURRENT-OBJECT))
                    (LOOK-AT-FIRST-IN-FILE
                         (FORGET-CURRENT-FILE
                          DESTROY-A-FILE
                          REMEMBER-CURRENT-FILE
                          FORGET-REMEMBERED-FILE))
            >>
        ]
        >>
    ]
    (PICK-OBJECT-FROM-FILE
         (CLOSE-EYES
          FIND-COLOR-OF-PIECE
          PIECE-HAS-STRAIGHT-EDGE
          PIECE-FITS-IN-PUZZLE
          FORGET-CURRENT-FILE
          DESTROY-A-FILE
          REMEMBER-CURRENT-FILE
          FORGET-REMEMBERED-FILE
          REMEMBER-CURRENT-OBJECT))
    (FIT-PIECE-IN-PUZZLE
         (CLOSE-EYES
          OBJECT-IN-HAND-IN-VIEW
          FIND-COLOR-OF-PIECE
          PIECE-HAS-STRAIGHT-EDGE
          PIECE-FITS-IN-PUZZLE
          PUT-OBJECT-IN-FILE
          FORGET-CURRENT-FILE
          DESTROY-A-FILE
          FILE-IS-EMPTY
          REMEMBER-CURRENT-FILE
          FORGET-REMEMBERED-FILE
          REMEMBER-CURRENT-OBJECT))
    (PIECE-PUT-IN-PUZZLE
         (CLOSE-EYES
          OBJECT-IN-HAND-IN-VIEW
          FIND-COLOR-OF-PIECE
          PIECE-HAS-STRAIGHT-EDGE
          PIECE-FITS-IN-PUZZLE
          FIT-PIECE-IN-PUZZLE
          PUT-OBJECT-IN-FILE
          FORGET-CURRENT-FILE
          DESTROY-A-FILE
          REMEMBER-CURRENT-FILE
          FORGET-REMEMBERED-FILE
          FILE-IS-EMPTY
          REMEMBER-CURRENT-OBJECT))
    [REPEAT    (WHILE  NIL)
```

```
                    (UNTIL  (PUZZLE-IS-FINISHED))
        <<
          (FORGET-CURRENT-FILE
                (FILE-IS-EMPTY
                 DESTROY-A-FILE
                 REMEMBER-CURRENT-FILE
                 FORGET-REMEMBERED-FILE))
          (PICK-A-FILE
                (FORGET-REMEMBERED-FILE))
          (LOOK-AT-FIRST-IN-FILE
                (FORGET-CURRENT-FILE
                 DESTROY-A-FILE
                 REMEMBER-CURRENT-FILE
                 REMEMBERED-FILE-IS-CURRENT
                 FORGET-REMEMBERED-FILE))
          (REMEMBER-CURRENT-OBJECT
                (CLOSE-EYES
                 FIND-COLOR-OF-PIECE
                 PICK-OBJECT-FROM-FILE
                 FORGET-CURRENT-FILE
                 DESTROY-A-FILE
                 LOOK-AT-NEXT-IN-FILE
                 REMEMBER-CURRENT-FILE
                 REMEMBERED-FILE-IS-CURRENT
                 FORGET-REMEMBERED-FILE))
          (FIND-COLOR-OF-PIECE
                (CLOSE-EYES
                 PICK-OBJECT-FROM-FILE
                 FORGET-CURRENT-FILE
                 DESTROY-A-FILE
                 LOOK-AT-NEXT-IN-FILE
                 REMEMBER-CURRENT-FILE
                 REMEMBERED-FILE-IS-CURRENT
                 FORGET-REMEMBERED-FILE
                 REMEMBER-CURRENT-OBJECT
                 FORGET-REMEMBERED-OBJECT
                 REMEMBERED-OBJECT-IN-VIEW))
          (LOOK-AT-NEXT-IN-FILE
                (CLOSE-EYES
                 FIND-COLOR-OF-PIECE
                 FORGET-COLOR-OF-PIECE
                 PIECE-HAS-CURRENT-COLOR
                 PICK-OBJECT-FROM-FILE
                 FORGET-CURRENT-FILE
                 DESTROY-A-FILE
                 REMEMBER-CURRENT-FILE
                 REMEMBERED-FILE-IS-CURRENT
                 FORGET-REMEMBERED-FILE
                 REMEMBER-CURRENT-OBJECT
                 FORGET-REMEMBERED-OBJECT
                 REMEMBERED-OBJECT-IN-VIEW))
        [REPEAT     (WHILE  NIL)
                    (UNTIL  (REMEMBERED-OBJECT-IN-VIEW))
            <<
            [REPEAT      (WHILE  NIL)
                         (UNTIL  (PIECE-HAS-CURRENT-COLOR))
                <<
                (LOOK-AT-NEXT-IN-FILE
                      (OR.*
                       (CLOSE-EYES
```

- 142 -

```
                              FIND-COLOR-OF-PIECE
                              FORGET-COLOR-OF-PIECE
                              PIECE-HAS-CURRENT-COLOR
                              PIECE-FITS-IN-PUZZLE
                              PICK-OBJECT-FROM-FILE
                              FORGET-CURRENT-FILE
                              DESTROY-A-FILE
                              REMEMBER-CURRENT-FILE
                              REMEMBERED-FILE-IS-CURRENT
                              FORGET-REMEMBERED-FILE
                              REMEMBER-CURRENT-OBJECT
                              FORGET-REMEMBERED-OBJECT
                              REMEMBERED-OBJECT-IN-VIEW)
                             (CLOSE-EYES
                              FIND-COLOR-OF-PIECE
                              PICK-OBJECT-FROM-FILE
                              FORGET-CURRENT-FILE
                              DESTROY-A-FILE
                              REMEMBER-CURRENT-FILE
                              REMEMBERED-FILE-IS-CURRENT
                              FORGET-REMEMBERED-FILE
                              REMEMBER-CURRENT-OBJECT
                              FORGET-REMEMBERED-OBJECT)))
            >>
     ]
     [REPEAT    (WHILE  (FORGET-COLOR-OF-PIECE
                         PIECE-HAS-CURRENT-COLOR))
                (UNTIL  (REMEMBERED-OBJECT-IN-VIEW))
         <<
       (PICK-OBJECT-FROM-FILE
            (OR.*
             (CLOSE-EYES
              FIND-COLOR-OF-PIECE
              FORGET-COLOR-OF-PIECE
              PIECE-HAS-CURRENT-COLOR
              PIECE-FITS-IN-PUZZLE
              FORGET-CURRENT-FILE
              DESTROY-A-FILE
              LOOK-AT-NEXT-IN-FILE
              REMEMBER-CURRENT-FILE
              REMEMBERED-FILE-IS-CURRENT
              FORGET-REMEMBERED-FILE
              REMEMBER-CURRENT-OBJECT
              FORGET-REMEMBERED-OBJECT)
             (CLOSE-EYES
              FIND-COLOR-OF-PIECE
              FORGET-COLOR-OF-PIECE
              PIECE-HAS-CURRENT-COLOR
              FORGET-CURRENT-FILE
              DESTROY-A-FILE
              LOOK-AT-NEXT-IN-FILE
              REMEMBER-CURRENT-FILE
              REMEMBERED-FILE-IS-CURRENT
              FORGET-REMEMBERED-FILE
              REMEMBER-CURRENT-OBJECT
              FORGET-REMEMBERED-OBJECT)))
       (FORGET-CURRENT-FILE
            (OR.*
             (CLOSE-EYES
              OBJECT-IN-HAND-IN-VIEW
```

- 143 -

```
                    FIND-COLOR-OF-PIECE
                    FORGET-COLOR-OF-PIECE
                    PIECE-HAS-CURRENT-COLOR
                    PIECE-FITS-IN-PUZZLE
                    FIT-PIECE-IN-PUZZLE
                    PUT-OBJECT-IN-FILE
                    DESTROY-A-FILE
                    LOOK-AT-FIRST-IN-FILE
                    REMEMBER-CURRENT-FILE
                    REMEMBERED-FILE-IS-CURRENT
                    FORGET-REMEMBERED-FILE
                    REMEMBER-CURRENT-OBJECT
                    FORGET-REMEMBERED-OBJECT)
                   (CLOSE-EYES
                    OBJECT-IN-HAND-IN-VIEW
                    FIND-COLOR-OF-PIECE
                    FORGET-COLOR-OF-PIECE
                    PIECE-HAS-CURRENT-COLOR
                    PUT-OBJECT-IN-FILE
                    DESTROY-A-FILE
                    LOOK-AT-FIRST-IN-FILE
                    REMEMBER-CURRENT-FILE
                    REMEMBERED-FILE-IS-CURRENT
                    FORGET-REMEMBERED-FILE
                    REMEMBER-CURRENT-OBJECT
                    FORGET-REMEMBERED-OBJECT)))
             (PICK-A-FILE
                   (OR.*
                   (CLOSE-EYES
                    OBJECT-IN-HAND-IN-VIEW
                    FIND-COLOR-OF-PIECE
                    FORGET-COLOR-OF-PIECE
                    PIECE-HAS-CURRENT-COLOR
                    PIECE-FITS-IN-PUZZLE
                    FIT-PIECE-IN-PUZZLE
                    MAKE-A-FILE
                    FORGET-REMEMBERED-FILE
                    REMEMBER-CURRENT-OBJECT
                    FORGET-REMEMBERED-OBJECT)
                   (CLOSE-EYES
                    OBJECT-IN-HAND-IN-VIEW
                    FIND-COLOR-OF-PIECE
                    FORGET-COLOR-OF-PIECE
                    PIECE-HAS-CURRENT-COLOR
                    MAKE-A-FILE
                    FORGET-REMEMBERED-FILE
                    REMEMBER-CURRENT-OBJECT
                    FORGET-REMEMBERED-OBJECT)))
             (PUT-OBJECT-IN-FILE
                   (OR.*
                   (CLOSE-EYES
                    OBJECT-IN-HAND-IN-VIEW
                    FIND-COLOR-OF-PIECE
                    FORGET-COLOR-OF-PIECE
                    PIECE-HAS-CURRENT-COLOR
                    FORGET-CURRENT-FILE
                    DESTROY-A-FILE
                    LOOK-AT-FIRST-IN-FILE
                    REMEMBER-CURRENT-FILE
                    FORGET-REMEMBERED-FILE
```

```
                              REMEMBER-CURRENT-OBJECT
                              FORGET-REMEMBERED-OBJECT)
                             (CLOSE-EYES
                              OBJECT-IN-HAND-IN-VIEW
                              FIND-COLOR-OF-PIECE
                              FORGET-COLOR-OF-PIECE
                              PIECE-HAS-CURRENT-COLOR
                              FORGET-CURRENT-FILE
                              FILE-IS-EMPTY
                              DESTROY-A-FILE
                              REMEMBER-CURRENT-FILE
                              FORGET-REMEMBERED-FILE
                              REMEMBER-CURRENT-OBJECT
                              FORGET-REMEMBERED-OBJECT)
                             (CLOSE-EYES
                              OBJECT-IN-HAND-IN-VIEW
                              FIND-COLOR-OF-PIECE
                              FORGET-COLOR-OF-PIECE
                              PIECE-HAS-CURRENT-COLOR
                              PIECE-FITS-IN-PUZZLE
                              FIT-PIECE-IN-PUZZLE
                              FORGET-CURRENT-FILE
                              DESTROY-A-FILE
                              LOOK-AT-FIRST-IN-FILE
                              REMEMBER-CURRENT-FILE
                              FORGET-REMEMBERED-FILE
                              REMEMBER-CURRENT-OBJECT
                              FORGET-REMEMBERED-OBJECT)))
                 (FORGET-CURRENT-FILE
                         (OR.*
                          (CLOSE-EYES
                           FIND-COLOR-OF-PIECE
                           FORGET-COLOR-OF-PIECE
                           PIECE-HAS-CURRENT-COLOR
                           DESTROY-A-FILE
                           REMEMBER-CURRENT-FILE
                           FORGET-REMEMBERED-FILE
                           REMEMBER-CURRENT-OBJECT
                           FORGET-REMEMBERED-OBJECT)
                          (CLOSE-EYES
                           FIND-COLOR-OF-PIECE
                           FORGET-COLOR-OF-PIECE
                           PIECE-HAS-CURRENT-COLOR
                           PIECE-FITS-IN-PUZZLE
                           DESTROY-A-FILE
                           LOOK-AT-FIRST-IN-FILE
                           REMEMBER-CURRENT-FILE
                           FORGET-REMEMBERED-FILE
                           REMEMBER-CURRENT-OBJECT
                           FORGET-REMEMBERED-OBJECT)))
                 (PICK-A-FILE
                         (OR.*
                          (CLOSE-EYES
                           FIND-COLOR-OF-PIECE
                           FORGET-COLOR-OF-PIECE
                           PIECE-HAS-CURRENT-COLOR
                           PIECE-FITS-IN-PUZZLE
                           FORGET-REMEMBERED-FILE
                           REMEMBER-CURRENT-OBJECT
                           FORGET-REMEMBERED-OBJECT)
```

- 145 -

```
                        (CLOSE-EYES
                         FIND-COLOR-OF-PIECE
                         FORGET-COLOR-OF-PIECE
                         PIECE-HAS-CURRENT-COLOR
                         FORGET-REMEMBERED-FILE
                         REMEMBER-CURRENT-OBJECT
                         FORGET-REMEMBERED-OBJECT)))
                    (LOOK-AT-FIRST-IN-FILE
                        (OR.*
                         (CLOSE-EYES
                          FIND-COLOR-OF-PIECE
                          FORGET-COLOR-OF-PIECE
                          PIECE-HAS-CURRENT-COLOR
                          PIECE-FITS-IN-PUZZLE
                          FORGET-CURRENT-FILE
                          DESTROY-A-FILE
                          REMEMBER-CURRENT-FILE
                          REMEMBERED-FILE-IS-CURRENT
                          FORGET-REMEMBERED-FILE
                          REMEMBER-CURRENT-OBJECT
                          FORGET-REMEMBERED-OBJECT)
                         (CLOSE-EYES
                          FIND-COLOR-OF-PIECE
                          FORGET-COLOR-OF-PIECE
                          PIECE-HAS-CURRENT-COLOR
                          FORGET-CURRENT-FILE
                          DESTROY-A-FILE
                          REMEMBER-CURRENT-FILE
                          REMEMBERED-FILE-IS-CURRENT
                          FORGET-REMEMBERED-FILE
                          REMEMBER-CURRENT-OBJECT
                          FORGET-REMEMBERED-OBJECT)))
                >>
            ]
            >>
        ]
        (FORGET-REMEMBERED-OBJECT
            (OR.*
             (CLOSE-EYES
              FIND-COLOR-OF-PIECE
              FORGET-COLOR-OF-PIECE
              PIECE-HAS-CURRENT-COLOR
              PICK-OBJECT-FROM-FILE
              FORGET-CURRENT-FILE
              DESTROY-A-FILE
              LOOK-AT-NEXT-IN-FILE
              REMEMBER-CURRENT-FILE
              REMEMBERED-FILE-IS-CURRENT
              FORGET-REMEMBERED-FILE
              REMEMBER-CURRENT-OBJECT
              REMEMBERED-OBJECT-IN-VIEW)
             (CLOSE-EYES
              FIND-COLOR-OF-PIECE
              FORGET-COLOR-OF-PIECE
              PIECE-HAS-CURRENT-COLOR
              PICK-OBJECT-FROM-FILE
              FORGET-CURRENT-FILE
              DESTROY-A-FILE
              REMEMBER-CURRENT-FILE
              REMEMBERED-FILE-IS-CURRENT
```

```
                              FORGET-REMEMBERED-FILE
                              REMEMBER-CURRENT-OBJECT
                              REMEMBERED-OBJECT-IN-VIEW)))
                   (PICK-OBJECT-FROM-FILE
                          (OR.*
                           (CLOSE-EYES
                            FIND-COLOR-OF-PIECE
                            FORGET-COLOR-OF-PIECE
                            PIECE-HAS-CURRENT-COLOR
                            PIECE-FITS-IN-PUZZLE
                            FORGET-CURRENT-FILE
                            DESTROY-A-FILE
                            LOOK-AT-NEXT-IN-FILE
                            REMEMBER-CURRENT-FILE
                            REMEMBERED-FILE-IS-CURRENT
                            FORGET-REMEMBERED-FILE
                            REMEMBER-CURRENT-OBJECT)
                           (CLOSE-EYES
                            FIND-COLOR-OF-PIECE
                            FORGET-COLOR-OF-PIECE
                            PIECE-HAS-CURRENT-COLOR
                            FORGET-CURRENT-FILE
                            DESTROY-A-FILE
                            REMEMBER-CURRENT-FILE
                            REMEMBERED-FILE-IS-CURRENT
                            FORGET-REMEMBERED-FILE
                            REMEMBER-CURRENT-OBJECT)
                           (CLOSE-EYES
                            FIND-COLOR-OF-PIECE
                            FORGET-COLOR-OF-PIECE
                            PIECE-HAS-CURRENT-COLOR
                            PIECE-FITS-IN-PUZZLE
                            FORGET-CURRENT-FILE
                            DESTROY-A-FILE
                            REMEMBER-CURRENT-FILE
                            REMEMBERED-FILE-IS-CURRENT
                            FORGET-REMEMBERED-FILE
                            REMEMBER-CURRENT-OBJECT)))
                   (FORGET-CURRENT-FILE
                          (OR.*
                           (CLOSE-EYES
                            OBJECT-IN-HAND-IN-VIEW
                            FIND-COLOR-OF-PIECE
                            FORGET-COLOR-OF-PIECE
                            PIECE-HAS-CURRENT-COLOR
                            PUT-OBJECT-IN-FILE
                            DESTROY-A-FILE
                            LOOK-AT-FIRST-IN-FILE
                            REMEMBER-CURRENT-FILE
                            REMEMBERED-FILE-IS-CURRENT
                            FORGET-REMEMBERED-FILE
                            REMEMBER-CURRENT-OBJECT)
                           (CLOSE-EYES
                            OBJECT-IN-HAND-IN-VIEW
                            FIND-COLOR-OF-PIECE
                            FORGET-COLOR-OF-PIECE
                            PIECE-HAS-CURRENT-COLOR
                            PIECE-FITS-IN-PUZZLE
                            FIT-PIECE-IN-PUZZLE
                            PUT-OBJECT-IN-FILE
```

```
                    DESTROY-A-FILE
                    LOOK-AT-FIRST-IN-FILE
                    REMEMBER-CURRENT-FILE
                    REMEMBERED-FILE-IS-CURRENT
                    FORGET-REMEMBERED-FILE
                    REMEMBER-CURRENT-OBJECT)
                   (CLOSE-EYES
                    OBJECT-IN-HAND-IN-VIEW
                    FIND-COLOR-OF-PIECE
                    FORGET-COLOR-OF-PIECE
                    PIECE-HAS-CURRENT-COLOR
                    PUT-OBJECT-IN-FILE
                    DESTROY-A-FILE
                    FILE-IS-EMPTY
                    REMEMBER-CURRENT-FILE
                    REMEMBERED-FILE-IS-CURRENT
                    FORGET-REMEMBERED-FILE
                    REMEMBER-CURRENT-OBJECT)))
              (PICK-A-FILE
                   (OR.*
                   (CLOSE-EYES
                    OBJECT-IN-HAND-IN-VIEW
                    FIND-COLOR-OF-PIECE
                    FORGET-COLOR-OF-PIECE
                    PIECE-HAS-CURRENT-COLOR
                    MAKE-A-FILE
                    FORGET-REMEMBERED-FILE
                    REMEMBER-CURRENT-OBJECT)
                   (CLOSE-EYES
                    OBJECT-IN-HAND-IN-VIEW
                    FIND-COLOR-OF-PIECE
                    FORGET-COLOR-OF-PIECE
                    PIECE-HAS-CURRENT-COLOR
                    MAKE-A-FILE
                    FORGET-REMEMBERED-FILE
                    PIECE-FITS-IN-PUZZLE
                    FIT-PIECE-IN-PUZZLE
                    REMEMBER-CURRENT-OBJECT)))
              (PUT-OBJECT-IN-FILE
                   (OR.*
                   (CLOSE-EYES
                    OBJECT-IN-HAND-IN-VIEW
                    FIND-COLOR-OF-PIECE
                    FORGET-COLOR-OF-PIECE
                    PIECE-HAS-CURRENT-COLOR
                    FORGET-CURRENT-FILE
                    DESTROY-A-FILE
                    LOOK-AT-FIRST-IN-FILE
                    REMEMBER-CURRENT-FILE
                    FORGET-REMEMBERED-FILE
                    REMEMBER-CURRENT-OBJECT)
                   (CLOSE-EYES
                    OBJECT-IN-HAND-IN-VIEW
                    FIND-COLOR-OF-PIECE
                    FORGET-COLOR-OF-PIECE
                    PIECE-HAS-CURRENT-COLOR
                    FORGET-CURRENT-FILE
                    DESTROY-A-FILE
                    PIECE-FITS-IN-PUZZLE
                    FIT-PIECE-IN-PUZZLE
```

```
                    LOOK-AT-FIRST-IN-FILE
                    REMEMBER-CURRENT-FILE
                    FORGET-REMEMBERED-FILE
                    REMEMBER-CURRENT-OBJECT)))
       [REPEAT    (WHILE   (LOOK-AT-FIRST-IN-FILE))
                  (UNTIL   (FILE-IS-EMPTY))
           <<
             (LOOK-AT-FIRST-IN-FILE
                 (OR.*
                   (CLOSE-EYES
                    FIND-COLOR-OF-PIECE
                    FORGET-COLOR-OF-PIECE
                    PIECE-HAS-CURRENT-COLOR
                    PIECE-FITS-IN-PUZZLE
                    FORGET-CURRENT-FILE
                    DESTROY-A-FILE
                    REMEMBER-CURRENT-FILE
                    FORGET-REMEMBERED-FILE
                    REMEMBER-CURRENT-OBJECT)
                   (FORGET-CURRENT-FILE
                    DESTROY-A-FILE
                    REMEMBER-CURRENT-FILE
                    FORGET-REMEMBERED-FILE)))
             (PICK-OBJECT-FROM-FILE
                 (OR.*
                   (CLOSE-EYES
                    FIND-COLOR-OF-PIECE
                    FORGET-COLOR-OF-PIECE
                    PIECE-HAS-CURRENT-COLOR
                    PIECE-FITS-IN-PUZZLE
                    FORGET-CURRENT-FILE
                    DESTROY-A-FILE
                    LOOK-AT-NEXT-IN-FILE
                    REMEMBER-CURRENT-FILE
                    FORGET-REMEMBERED-FILE
                    REMEMBER-CURRENT-OBJECT)
                   (CLOSE-EYES
                    FIND-COLOR-OF-PIECE
                    FORGET-COLOR-OF-PIECE
                    PIECE-HAS-CURRENT-COLOR
                    PIECE-FITS-IN-PUZZLE
                    FORGET-CURRENT-FILE
                    DESTROY-A-FILE
                    REMEMBER-CURRENT-FILE
                    FORGET-REMEMBERED-FILE
                    REMEMBER-CURRENT-OBJECT)))
             (FIT-PIECE-IN-PUZZLE
                 (OR.*
                   (CLOSE-EYES
                    OBJECT-IN-HAND-IN-VIEW
                    FIND-COLOR-OF-PIECE
                    FORGET-COLOR-OF-PIECE
                    PIECE-HAS-CURRENT-COLOR
                    PIECE-FITS-IN-PUZZLE
                    PUT-OBJECT-IN-PILE
                    FORGET-CURRENT-FILE
                    DESTROY-A-FILE
                    LOOK-AT-FIRST-IN-FILE
                    REMEMBER-CURRENT-FILE
                    FORGET-REMEMBERED-FILE
```

- 149 -

```
                                REMEMBER-CURRENT-OBJECT)
                              (CLOSE-EYES
                               OBJECT-IN-HAND-IN-VIEW
                               FIND-COLOR-OF-PIECE
                               FORGET-COLOR-OF-PIECE
                               PIECE-HAS-CURRENT-COLOR
                               PIECE-FITS-IN-PUZZLE
                               PUT-OBJECT-IN-FILE
                               FORGET-CURRENT-FILE
                               FILE-IS-EMPTY
                               DESTROY-A-FILE
                               REMEMBER-CURRENT-FILE
                               FORGET-REMEMBERED-FILE
                               REMEMBER-CURRENT-OBJECT)))
                        (PIECE-PUT-IN-PUZZLE
                              (OR.*
                              (CLOSE-EYES
                               OBJECT-IN-HAND-IN-VIEW
                               FIND-COLOR-OF-PIECE
                               FORGET-COLOR-OF-PIECE
                               PIECE-HAS-CURRENT-COLOR
                               PIECE-FITS-IN-PUZZLE
                               FIT-PIECE-IN-PUZZLE
                               PUT-OBJECT-IN-FILE
                               FORGET-CURRENT-FILE
                               DESTROY-A-FILE
                               LOOK-AT-FIRST-IN-FILE
                               REMEMBER-CURRENT-FILE
                               FORGET-REMEMBERED-FILE
                               REMEMBER-CURRENT-OBJECT)
                              (CLOSE-EYES
                               OBJECT-IN-HAND-IN-VIEW
                               FIND-COLOR-OF-PIECE
                               FORGET-COLOR-OF-PIECE
                               PIECE-HAS-CURRENT-COLOR
                               PIECE-FITS-IN-PUZZLE
                               FIT-PIECE-IN-PUZZLE
                               PUT-OBJECT-IN-FILE
                               FORGET-CURRENT-FILE
                               FILE-IS-EMPTY
                               DESTROY-A-FILE
                               REMEMBER-CURRENT-FILE
                               FORGET-REMEMBERED-FILE
                               REMEMBER-CURRENT-OBJECT)))
                    >>
           ]
             >>
         ]
         (FORGET-CURRENT-FILE
              (FILE-IS-EMPTY
              DESTROY-A-FILE
              REMEMBER-CURRENT-FILE
              PUZZLE-IS-FINISHED
              FORGET-REMEMBERED-FILE))
         [REPEAT     (WHILE  (PICK-A-FILE FORGET-REMEMBERED-FILE))
                     (UNTIL  (THERE-ARE-NO-FILES))
             <<
             (PICK-A-FILE
                  (PUZZLE-IS-FINISHED
                  FORGET-REMEMBERED-FILE))
```

```
        (DESTROY-A-FILE
                (PUZZLE-IS-FINISHED
                FORGET-CURRENT-FILE
                PILE-IS-EMPTY
                REMEMBER-CURRENT-FILE
                FORGET-REMEMBERED-FILE))
    >>
  )
  (PUZZLE-IS-FINISHED
        (THERE-ARE-NO-FILES))
>>
```

APPENDIX 3
==========

FOURTH PUZZLE EXPERIMENT

?*** UT LISP - VER 4.1 (78/04/10).
(*** PROSYS SUBSYSTEM 1.1 (79/04/15))
(*** CRAPS CONTROL ***)

(*** EXPERIMENT 4 SPACE SHIP PUZZLE ***)

INITIAL WORKING MEMORY
((NUMBER-OF-PIECES 24)
 (NUMBER-IN-HEAP 24)
 (NUMBER-OF-PILES 0)
 (IN-PUZZLE NOTHING)
 (CURRENT-COLOR NOTHING)
 (HOLDING NOTHING)
 (LOOKING-AT NOTHING)
 (CURRENT-FILE NONE)
 (REMEMBERED-FILE NONE)
 (REMEMBERED-OBJECT NONE)
 (ALL-FILES)
 (IN-HEAP (PIECE 22 RED))
 (IN-HEAP (PIECE 23 BLACK))
 (IN-HEAP (PIECE 12 BLACK))
 (IN-HEAP (PIECE 13 BLACK))
 (IN-HEAP (PIECE 14 BLACK))
 (IN-HEAP (PIECE 15 BLACK))
 (IN-HEAP (PIECE 24 RED))
 (IN-HEAP (PIECE 25 BLACK))
 (IN-HEAP (PIECE 32 BLACK))
 (IN-HEAP (PIECE 33 RED))
 (IN-HEAP (PIECE 34 SILVER))
 (IN-HEAP (PIECE 35 SILVER))
 (IN-HEAP (PIECE 42 BLACK))
 (IN-HEAP (PIECE 43 BLACK))
 (IN-HEAP (PIECE 44 SILVER))
 (IN-HEAP (PIECE 45 BLACK))
 (IN-HEAP (PIECE 52 BROWN))
 (IN-HEAP (PIECE 53 BROWN))
 (IN-HEAP (PIECE 55 BLACK))
 (IN-HEAP (PIECE 62 BROWN))
 (IN-HEAP (PIECE 63 BROWN))
 (IN-HEAP (PIECE 64 BROWN))
 (IN-HEAP (PIECE 54 BROWN))
 (IN-HEAP (PIECE 65 BLACK))
 (L 22 0 RED)
 (L 23 29 BLACK)
 (T 23 30 BLACK)
 (R 23 -36 BLACK)
 (B 23 -40 BLACK)
 (L 12 0 BLACK)
 (T 12 0 BLACK)
 (R 12 23 BLACK)
 (R 13 -24 BLACK)
 (B 13 -30 BLACK)
 (L 14 24 BLACK)

```
(T 14 0 BLACK)
(R 14 31 BLACK)
(R 15 0 BLACK)
(B 15 38 BLACK)
(L 24 36 RED)
(T 24 42 BLACK)
(R 24 -44 BLACK)
(L 65 108 BROWN)
(T 65 104 BLACK)
(B 24 46 RED)
(L 25 44 BLACK)
(T 25 -38 BLACK)
(R 25 0 BLACK)
(B 25 47 BLACK)
(L 32 0 BLACK)
(T 32 -35 BLACK)
(R 32 52 BLACK)
(B 32 -53 BLACK)
(L 33 -52 BLACK)
(T 33 40 RED)
(R 33 54 RED)
(B 33 -55 BLACK)
(L 34 -54 RED)
(T 34 -46 BLACK)
(R 34 56 SILVER)
(B 34 -57 SILVER)
(L 35 -56 SILVER)
(T 35 -47 SILVER)
(R 35 0 BLACK)
(B 35 60 SILVER)
(L 42 0 L-BLACK)
(T 42 53 BLACK)
(R 42 -68 BLACK)
(T 44 57 SILVER)
(L 64 107 BROWN)
(T 64 -103 BROWN)
(R 44 76 SILVER)
(B 44 -77 SILVER)
(L 45 -76 BLACK)
(T 45 -60 BLACK)
(R 45 0 BLACK)
(B 45 82 BLACK)
(L 52 0 BROWN)
(T 52 -69 BLACK)
(R 52 -86 BROWN)
(B 52 101 BROWN)
(L 53 86 BROWN)
(T 53 74 YELLOW)
(B 12 -27 BLACK)
(L 13 -23 BLACK)
(T 13 0 BLACK)
(R 53 87 BROWN)
(B 53 -102 BROWN)
(L 55 88 BLACK)
(T 55 82 BLACK)
(R 55 0 BLACK)
(B 42 69 BLACK)
(L 43 68 BLACK)
(T 43 55 BLACK)
(R 43 -62 SILVER)
```

```
(B 43 -64 BLACK)
(L 44 72 SILVER)
(B 55 -104 BLACK)
(T 22 27 BLACK)
(R 22 -29 RED)
(B 22 35 RED)
(L 62 0 BROWN)
(T 62 -101 BROWN)
(R 62 106 BROWN)
(B 62 0 BROWN)
(L 63 -106 BROWN)
(T 63 102 BROWN)
(R 63 -107 BROWN)
(B 63 0 BROWN)
(R 64 -108 BROWN)
(B 64 0 BROWN)
(B 14 -42 BLACK)
(L 15 -31 BLACK)
(T 15 0 BLACK)
(L 54 -87 BROWN)
(T 54 77 SILVER)
(R 54 -88 BROWN)
(B 54 103 BROWN)
(R 65 0 BLACK)
(B 65 0 BLACK)
```

```
*** BEGIN RUN ***

1. LOOK-AT-PIECE-IN-HEAP
2. PICK-UP-OBJECT-IN-VIEW
3. MAKE-A-FILE
4. LOOK-AT-PIECE-IN-HEAP
5. PICK-UP-OBJECT-IN-VIEW
6. PUT-OBJECT-IN-FILE
7. LOOK-AT-PIECE-IN-HEAP
8. PICK-UP-OBJECT-IN-VIEW
9. PUT-OBJECT-IN-FILE

10. LOOK-AT-PIECE-IN-HEAP
11. PICK-UP-OBJECT-IN-VIEW
12. PUT-OBJECT-IN-FILE
13. LOOK-AT-PIECE-IN-HEAP
14. PICK-UP-OBJECT-IN-VIEW
15. PUT-OBJECT-IN-FILE
16. LOOK-AT-PIECE-IN-HEAP
17. PICK-UP-OBJECT-IN-VIEW
18. PUT-OBJECT-IN-FILE
19. LOOK-AT-PIECE-IN-HEAP

20. PICK-UP-OBJECT-IN-VIEW
21. PUT-OBJECT-IN-FILE
22. LOOK-AT-PIECE-IN-HEAP
23. PICK-UP-OBJECT-IN-VIEW
24. PUT-OBJECT-IN-FILE
25. LOOK-AT-PIECE-IN-HEAP
26. PICK-UP-OBJECT-IN-VIEW
27. PUT-OBJECT-IN-FILE
28. LOOK-AT-PIECE-IN-HEAP
29. PICK-UP-OBJECT-IN-VIEW

30. PUT-OBJECT-IN-FILE
31. LOOK-AT-PIECE-IN-HEAP
32. PICK-UP-OBJECT-IN-VIEW
33. PUT-OBJECT-IN-FILE
34. LOOK-AT-PIECE-IN-HEAP
35. PICK-UP-OBJECT-IN-VIEW
36. PUT-OBJECT-IN-FILE
37. LOOK-AT-PIECE-IN-HEAP
38. PICK-UP-OBJECT-IN-VIEW
39. PUT-OBJECT-IN-FILE

40. LOOK-AT-PIECE-IN-HEAP
41. PICK-UP-OBJECT-IN-VIEW
42. PUT-OBJECT-IN-FILE
43. LOOK-AT-PIECE-IN-HEAP
44. PICK-UP-OBJECT-IN-VIEW
45. PUT-OBJECT-IN-FILE
46. LOOK-AT-PIECE-IN-HEAP
47. PICK-UP-OBJECT-IN-VIEW
48. PUT-OBJECT-IN-FILE
49. LOOK-AT-PIECE-IN-HEAP

50. PICK-UP-OBJECT-IN-VIEW
51. PUT-OBJECT-IN-FILE
52. LOOK-AT-PIECE-IN-HEAP
```

```
53.  PICK-UP-OBJECT-IN-VIEW
54.  PUT-OBJECT-IN-FILE
55.  LOOK-AT-PIECE-IN-HEAP
56.  PICK-UP-OBJECT-IN-VIEW
57.  PUT-OBJECT-IN-FILE
58.  LOOK-AT-PIECE-IN-HEAP
59.  PICK-UP-OBJECT-IN-VIEW

60.  PUT-OBJECT-IN-FILE
61.  LOOK-AT-PIECE-IN-HEAP
62.  PICK-UP-OBJECT-IN-VIEW
63.  PUT-OBJECT-IN-FILE
64.  LOOK-AT-PIECE-IN-HEAP
65.  PICK-UP-OBJECT-IN-VIEW
66.  PUT-OBJECT-IN-FILE
67.  LOOK-AT-PIECE-IN-HEAP
68.  PICK-UP-OBJECT-IN-VIEW
69.  PUT-OBJECT-IN-FILE

70.  LOOK-AT-PIECE-IN-HEAP

      WORKING MEMORY
((NUMBER-OF-PIECES 24)
 (NUMBER-IN-HEAP 1)
 (NUMBER-OF-FILES 1)
 (IN-PUZZLE NOTHING)
 (CURRENT-COLOR NOTHING)
 (HOLDING NOTHING)
 (LOOKING-AT (PIECE 22 RED))
 (CURRENT-FILE 1)
 (REMEMBERED-FILE NONE)
 (REMEMBERED-OBJECT NONE)
 (ALL-FILES 1)
 (IN-HEAP (PIECE 22 RED))
 (FILE 1
        (PIECE 65 BLACK)
        (PIECE 54 BROWN)
        (PIECE 64 BROWN)
        (PIECE 63 BROWN)
        (PIECE 62 BROWN)
        (PIECE 55 BLACK)
        (PIECE 53 BROWN)
        (PIECE 52 BROWN)
        (PIECE 45 BLACK)
        (PIECE 44 SILVER)
        (PIECE 43 BLACK)
        (PIECE 42 BLACK)
        (PIECE 35 SILVER)
        (PIECE 34 SILVER)
        (PIECE 33 RED)
        (PIECE 32 BLACK)
        (PIECE 25 BLACK)
        (PIECE 24 RED)
        (PIECE 15 BLACK)
        (PIECE 14 BLACK)
        (PIECE 13 BLACK)
        (PIECE 12 BLACK)
        (PIECE 23 BLACK)))
71.  PICK-UP-OBJECT-IN-VIEW
72.  PUT-OBJECT-IN-FILE
```

```
                          73. EMPTY-HEAP
                          74. REMEMBER-CURRENT-FILE
                          75. LOOK-AT-FIRST-IN-FILE
                          76. REMEMBER-CURRENT-OBJECT
                          77. LOOK-AT-NEXT-IN-FILE


**META-RULES-CALLED**

                        UNIT
          (PICK-OBJECT-FROM-FILE
           (CLOSE-EYES FIND-COLOR-OF-PIECE ...))

                        WANT ACTIVE
               PIECE-HAS-STRAIGHT-EDGE

                        WANT INACTIVE
               PICK-OBJECT-FROM-FILE

                        CURRENTLY ACTIVE
               PICK-OBJECT-FROM-FILE
               LOOK-AT-NEXT-IN-FILE
               REMEMBER-CURRENT-OBJECT
               FIND-COLOR-OF-PIECE
               CLOSE-EYES
               DESTROY-A-FILE
               FORGET-REMEMBERED-OBJECT
               REMEMBERED-FILE-IS-CURRENT
               REMEMBER-CURRENT-FILE
               FORGET-REMEMBERED-FILE
               FORGET-CURRENT-FILE

                        TRY LIST
               PICK-OBJECT-FROM-FILE
               LOOK-AT-NEXT-IN-FILE
               LOOK-AT-NEXT-IN-FILE
               PICK-OBJECT-FROM-FILE
               LOOK-AT-NEXT-IN-FILE



                      78. LOOK-AT-NEXT-IN-FILE
                      79. PICK-OBJECT-FROM-FILE

                      80. FORGET-CURRENT-FILE

                      WORKING MEMORY
               ((NUMBER-OF-PIECES 24)
                (NUMBER-OF-FILES 1)
                (IN-PUZZLE NOTHING)
                (CURRENT-COLOR NOTHING)
                (HOLDING (PIECE 64 BROWN))
                (LOOKING-AT (PIECE 64 BROWN))
                (CURRENT-FILE NONE)
                (REMEMBERED-FILE 1)
                (REMEMBERED-OBJECT (PIECE 65 BLACK))
                (ALL-FILES 1)
                (FILE 1
                      (PIECE 63 BROWN)
```

- 157 -

```
                        (PIECE 62 BROWN)
                        (PIECE 55 BLACK)
                        (PIECE 53 BROWN)
                        (PIECE 52 BROWN)
                        (PIECE 45 BLACK)
                        (PIECE 44 SILVER)
                        (PIECE 43 BLACK)
                        (PIECE 42 BLACK)
                        (PIECE 35 SILVER)
                        (PIECE 34 SILVER)
                        (PIECE 33 RED)
                        (PIECE 32 BLACK)
                        (PIECE 25 BLACK)
                        (PIECE 24 RED)
                        (PIECE 15 BLACK)
                        (PIECE 14 BLACK)
                        (PIECE 13 BLACK)
                        (PIECE 12 BLACK)
                        (PIECE 23 BLACK)
                        (PIECE 22 RED)
                        (PIECE 65 BLACK)
                        (PIECE 54 BROWN)))
           81.  MAKE-A-FILE
           82.  FORGET-CURRENT-FILE
           83.  PICK-A-FILE
           84.  LOOK-AT-FIRST-IN-FILE
           85.  PICK-OBJECT-FROM-FILE
           86.  FORGET-CURRENT-FILE
           87.  PICK-A-FILE
           88.  PUT-OBJECT-IN-FILE
           89.  FORGET-CURRENT-FILE

           90.  PICK-A-FILE
           91.  LOOK-AT-FIRST-IN-FILE
           92.  PICK-OBJECT-FROM-FILE
           93.  FORGET-CURRENT-FILE
           94.  PICK-A-FILE
           95.  PUT-OBJECT-IN-FILE
           96.  FORGET-CURRENT-FILE
           97.  PICK-A-FILE
           98.  LOOK-AT-FIRST-IN-FILE
           99.  PICK-OBJECT-FROM-FILE

           100. FORGET-CURRENT-FILE
           101. PICK-A-FILE
           102. PUT-OBJECT-IN-FILE
           103. FORGET-CURRENT-FILE
           104. PICK-A-FILE
           105. LOOK-AT-FIRST-IN-FILE
           106. LOOK-AT-NEXT-IN-FILE
           107. PICK-OBJECT-FROM-FILE
           108. FORGET-CURRENT-FILE
           109. PICK-A-FILE

           110. PUT-OBJECT-IN-FILE
           111. FORGET-CURRENT-FILE
           112. PICK-A-FILE
           113. LOOK-AT-FIRST-IN-FILE
           114. PICK-OBJECT-FROM-FILE
           115. FORGET-CURRENT-FILE
```

```
116. PICK-A-FILE
117. PUT-OBJECT-IN-FILE
118. FORGET-CURRENT-FILE
119. PICK-A-FILE

120. LOOK-AT-FIRST-IN-FILE
121. LOOK-AT-NEXT-IN-FILE
122. LOOK-AT-NEXT-IN-FILE
123. PICK-OBJECT-FROM-FILE
124. FORGET-CURRENT-FILE
125. PICK-A-FILE
126. PUT-OBJECT-IN-FILE
127. FORGET-CURRENT-FILE
128. PICK-A-FILE
129. LOOK-AT-FIRST-IN-FILE

130. PICK-OBJECT-FROM-FILE
131. FORGET-CURRENT-FILE
132. PICK-A-FILE
133. PUT-OBJECT-IN-FILE
134. FORGET-CURRENT-FILE
135. PICK-A-FILE
136. LOOK-AT-FIRST-IN-FILE
137. LOOK-AT-NEXT-IN-FILE
138. LOOK-AT-NEXT-IN-FILE
139. PICK-OBJECT-FROM-FILE

140. FORGET-CURRENT-FILE
141. PICK-A-FILE
142. PUT-OBJECT-IN-FILE
143. FORGET-CURRENT-FILE
144. PICK-A-FILE
145. LOOK-AT-FIRST-IN-FILE
146. PICK-OBJECT-FROM-FILE
147. FORGET-CURRENT-FILE
148. PICK-A-FILE
149. PUT-OBJECT-IN-FILE

150. FORGET-CURRENT-FILE
151. PICK-A-FILE
152. LOOK-AT-FIRST-IN-FILE
153. LOOK-AT-NEXT-IN-FILE
154. PICK-OBJECT-FROM-FILE
155. FORGET-CURRENT-FILE
156. PICK-A-FILE
157. PUT-OBJECT-IN-FILE
158. FORGET-CURRENT-FILE
159. PICK-A-FILE

160. LOOK-AT-FIRST-IN-FILE
161. PICK-OBJECT-FROM-FILE
162. FORGET-CURRENT-FILE
163. PICK-A-FILE
164. PUT-OBJECT-IN-FILE
165. FORGET-CURRENT-FILE
166. PICK-A-FILE
167. LOOK-AT-FIRST-IN-FILE
168. PICK-OBJECT-FROM-FILE
169. FORGET-CURRENT-FILE
```

```
170. PICK-A-FILE
171. PUT-OBJECT-IN-FILE
172. FORGET-CURRENT-FILE
173. PICK-A-FILE
174. LOOK-AT-FIRST-IN-FILE
175. PICK-OBJECT-FROM-FILE
176. FORGET-CURRENT-FILE
177. PICK-A-FILE
178. PUT-OBJECT-IN-FILE
179. FORGET-CURRENT-FILE

180. PICK-A-FILE
181. LOOK-AT-FIRST-IN-FILE
182. LOOK-AT-NEXT-IN-FILE
183. PICK-OBJECT-FROM-FILE
184. FORGET-CURRENT-FILE
185. PICK-A-FILE
186. PUT-OBJECT-IN-FILE
187. FORGET-CURRENT-FILE
188. PICK-A-FILE
189. LOOK-AT-FIRST-IN-FILE

190. PICK-OBJECT-FROM-FILE
191. FORGET-CURRENT-FILE
192. PICK-A-FILE
193. PUT-OBJECT-IN-FILE
194. FORGET-REMEMBERED-OBJECT
195. LOOK-AT-FIRST-IN-FILE
196. PICK-OBJECT-FROM-FILE
197. START-PUZZLE
198. LOOK-AT-FIRST-IN-FILE
199. PICK-OBJECT-FROM-FILE

200. FIT-PIECE-IN-PUZZLE
```

```
        WORKING MEMORY
((NUMBER-OF-PIECES 24)
 (NUMBER-OF-FILES 2)
 (IN-PUZZLE (PIECE 64 BROWN))
 (CURRENT-COLOR NOTHING)
 (HOLDING (PIECE 63 BROWN))
 (LOOKING-AT (PIECE 63 BROWN))
 (CURRENT-FILE 2)
 (REMEMBERED-FILE 1)
 (REMEMBERED-OBJECT NONE)
 (ALL-FILES 2 1)
 (FILE 1
        (PIECE 54 BROWN)
        (PIECE 53 BROWN)
        (PIECE 44 SILVER)
        (PIECE 43 BLACK)
        (PIECE 34 SILVER)
        (PIECE 33 RED)
        (PIECE 24 RED)
        (PIECE 23 BLACK))
 (FILE 2
        (PIECE 62 BROWN)
        (PIECE 55 BLACK)
        (PIECE 52 BROWN)
        (PIECE 45 BLACK)
```

```
                    (PIECE 42 BLACK)
                    (PIECE 35 SILVER)
                    (PIECE 32 BLACK)
                    (PIECE 25 BLACK)
                    (PIECE 15 BLACK)
                    (PIECE 14 BLACK)
                    (PIECE 13 BLACK)
                    (PIECE 12 BLACK)
                    (PIECE 22 RED)
                    (PIECE 65 BLACK))
            (NUMBER-IN-PUZZLE 1)
            (JOINED ((L 64 107 BROWN) (R 63 -107 BROWN)))
            (BEING-PUT-IN-PUZZLE (PIECE 63 BROWN)))

            201. PIECE-PUT-IN-PUZZLE
            202. LOOK-AT-FIRST-IN-PILE
            203. PICK-OBJECT-FROM-PILE
            204. FIT-PIECE-IN-PUZZLE
            205. PIECE-PUT-IN-PUZZLE
            206. LOOK-AT-FIRST-IN-PILE
            207. LOOK-AT-NEXT-IN-PILE
            208. PICK-OBJECT-FROM-PILE
            209. FIT-PIECE-IN-PUZZLE

            210. PIECE-PUT-IN-PUZZLE
            211. LOOK-AT-FIRST-IN-PILE
            212. LOOK-AT-NEXT-IN-PILE
            213. PICK-OBJECT-FROM-PILE
            214. FIT-PIECE-IN-PUZZLE
            215. PIECE-PUT-IN-PUZZLE
            216. LOOK-AT-FIRST-IN-PILE
            217. LOOK-AT-NEXT-IN-PILE
            218. PICK-OBJECT-FROM-PILE
            219. FIT-PIECE-IN-PUZZLE

            220. PIECE-PUT-IN-PUZZLE
            221. LOOK-AT-FIRST-IN-PILE
            222. LOOK-AT-NEXT-IN-PILE
            223. LOOK-AT-NEXT-IN-PILE
            224. LOOK-AT-NEXT-IN-PILE
            225. LOOK-AT-NEXT-IN-PILE
            226. LOOK-AT-NEXT-IN-PILE
            227. PICK-OBJECT-FROM-PILE
            228. FIT-PIECE-IN-PUZZLE
            229. PIECE-PUT-IN-PUZZLE

            230. LOOK-AT-FIRST-IN-PILE
            231. PICK-OBJECT-FROM-PILE
            232. FIT-PIECE-IN-PUZZLE
            233. PIECE-PUT-IN-PUZZLE
            234. LOOK-AT-FIRST-IN-PILE
            235. PICK-OBJECT-FROM-PILE
            236. FIT-PIECE-IN-PUZZLE
            237. PIECE-PUT-IN-PUZZLE
            238. LOOK-AT-FIRST-IN-PILE
            239. LOOK-AT-NEXT-IN-PILE

            240. LOOK-AT-NEXT-IN-PILE
            241. LOOK-AT-NEXT-IN-PILE
            242. LOOK-AT-NEXT-IN-PILE
```

```
243. LOOK-AT-NEXT-IN-FILE
244. LOOK-AT-NEXT-IN-FILE
245. PICK-OBJECT-FROM-FILE
246. FIT-PIECE-IN-PUZZLE
247. PIECE-PUT-IN-PUZZLE
248. LOOK-AT-FIRST-IN-FILE
249. LOOK-AT-NEXT-IN-FILE

250. LOOK-AT-NEXT-IN-FILE
251. LOOK-AT-NEXT-IN-FILE
252. LOOK-AT-NEXT-IN-FILE
253. LOOK-AT-NEXT-IN-FILE
254. PICK-OBJECT-FROM-FILE
255. FIT-PIECE-IN-PUZZLE
256. PIECE-PUT-IN-PUZZLE
257. LOOK-AT-FIRST-IN-FILE
258. LOOK-AT-NEXT-IN-FILE
259. LOOK-AT-NEXT-IN-FILE

260. LOOK-AT-NEXT-IN-FILE
261. LOOK-AT-NEXT-IN-FILE
262. PICK-OBJECT-FROM-FILE
263. FIT-PIECE-IN-PUZZLE
264. PIECE-PUT-IN-PUZZLE
265. LOOK-AT-FIRST-IN-FILE
266. LOOK-AT-NEXT-IN-FILE
267. LOOK-AT-NEXT-IN-FILE
268. LOOK-AT-NEXT-IN-FILE
269. PICK-OBJECT-FROM-FILE

270. FIT-PIECE-IN-PUZZLE
271. PIECE-PUT-IN-PUZZLE
272. LOOK-AT-FIRST-IN-FILE
273. LOOK-AT-NEXT-IN-FILE
274. LOOK-AT-NEXT-IN-FILE
275. PICK-OBJECT-FROM-FILE
276. FIT-PIECE-IN-PUZZLE
277. PIECE-PUT-IN-PUZZLE
278. LOOK-AT-FIRST-IN-FILE
279. LOOK-AT-NEXT-IN-FILE

280. PICK-OBJECT-FROM-FILE
281. FIT-PIECE-IN-PUZZLE
282. PIECE-PUT-IN-PUZZLE
283. LOOK-AT-FIRST-IN-FILE
284. PICK-OBJECT-FROM-FILE
285. FIT-PIECE-IN-PUZZLE
286. PIECE-PUT-IN-PUZZLE
287. FORGET-CURRENT-FILE
288. PICK-A-FILE
289. LOOK-AT-FIRST-IN-FILE

290. REMEMBER-CURRENT-OBJECT

        WORKING MEMORY
((NUMBER-OF-PIECES 24)
 (NUMBER-OF-FILES 2)
 (IN-PUZZLE (PIECE 64 BROWN))
 (IN-PUZZLE (PIECE 63 BROWN))
 (IN-PUZZLE (PIECE 62 BROWN))
```

```
(IN-PUZZLE (PIECE 52 BROWN))
(IN-PUZZLE (PIECE 42 BLACK))
(IN-PUZZLE (PIECE 32 BLACK))
(IN-PUZZLE (PIECE 22 RED))
(IN-PUZZLE (PIECE 65 BLACK))
(IN-PUZZLE (PIECE 55 BLACK))
(IN-PUZZLE (PIECE 12 BLACK))
(IN-PUZZLE (PIECE 13 BLACK))
(IN-PUZZLE (PIECE 14 BLACK))
(IN-PUZZLE (PIECE 15 BLACK))
(IN-PUZZLE (PIECE 25 BLACK))
(IN-PUZZLE (PIECE 35 SILVER))
(IN-PUZZLE (PIECE 45 BLACK))
(CURRENT-COLOR NOTHING)
(HOLDING NOTHING)
(LOOKING-AT (PIECE 54 BROWN))
(CURRENT-FILE 1)
(REMEMBERED-FILE 1)
(REMEMBERED-OBJECT (PIECE 54 BROWN))
(ALL-FILES 1 2)
(FILE 1
        (PIECE 54 BROWN)
        (PIECE 53 BROWN)
        (PIECE 44 SILVER)
        (PIECE 43 BLACK)
        (PIECE 34 SILVER)
        (PIECE 33 RED)
        (PIECE 24 RED)
        (PIECE 23 BLACK))
(FILE 2)
(NUMBER-IN-PUZZLE 16)
(JOINED ((L 64 107 BROWN) (R 63 -107 BROWN))
        ((L 63 -106 BROWN) (R 62 106 BROWN))
        ((T 62 -101 BROWN) (B 52 101 BROWN))
        ((T 52 -69 BLACK) (B 42 69 BLACK))
        ((T 42 53 BLACK) (B 32 -53 BLACK))
        ((T 32 -35 BLACK) (B 22 35 RED))
        ((R 64 -108 BROWN) (L 65 108 BROWN))
        ((T 65 104 BLACK) (B 55 -104 BLACK))
        ((T 22 27 BLACK) (B 12 -27 BLACK))
        ((R 12 23 BLACK) (L 13 -23 BLACK))
        ((R 13 -24 BLACK) (L 14 24 BLACK))
        ((R 14 31 BLACK) (L 15 -31 BLACK))
        ((B 15 38 BLACK) (T 25 -38 BLACK))
        ((B 25 47 BLACK) (T 35 -47 SILVER))
        ((B 35 60 SILVER) (T 45 -60 BLACK))))
    291. FIND-COLOR-OF-PIECE
    292. LOOK-AT-NEXT-IN-FILE
    293. PICK-OBJECT-FROM-FILE
    294. FORGET-CURRENT-FILE
    295. PICK-A-FILE
    296. PUT-OBJECT-IN-FILE
    297. FORGET-CURRENT-FILE
    298. PICK-A-FILE
    299. LOOK-AT-FIRST-IN-FILE

    300. LOOK-AT-NEXT-IN-FILE
    301. LOOK-AT-NEXT-IN-FILE
    302. LOOK-AT-NEXT-IN-FILE
    303. LOOK-AT-NEXT-IN-FILE
```

```
304. LOOK-AT-NEXT-IN-FILE
305. LOOK-AT-NEXT-IN-FILE
306. FORGET-REMEMBERED-OBJECT
307. PICK-OBJECT-FROM-FILE
308. FORGET-CURRENT-FILE
309. PICK-A-FILE

310. PUT-OBJECT-IN-FILE
311. LOOK-AT-FIRST-IN-FILE
312. PICK-OBJECT-FROM-FILE
313. FIT-PIECE-IN-PUZZLE
314. PIECE-PUT-IN-PUZZLE
315. LOOK-AT-FIRST-IN-FILE
316. PICK-OBJECT-FROM-FILE
317. FIT-PIECE-IN-PUZZLE
318. PIECE-PUT-IN-PUZZLE
319. FORGET-CURRENT-FILE

320. PICK-A-FILE
321. LOOK-AT-FIRST-IN-FILE
322. REMEMBER-CURRENT-OBJECT
323. FIND-COLOR-OF-PIECE
324. LOOK-AT-NEXT-IN-FILE
325. LOOK-AT-NEXT-IN-FILE
326. PICK-OBJECT-FROM-FILE
327. FORGET-CURRENT-FILE
328. PICK-A-FILE
329. PUT-OBJECT-IN-FILE

330. FORGET-CURRENT-FILE
331. PICK-A-FILE
332. LOOK-AT-FIRST-IN-FILE
333. LOOK-AT-NEXT-IN-FILE
334. LOOK-AT-NEXT-IN-FILE
335. LOOK-AT-NEXT-IN-FILE
336. FORGET-REMEMBERED-OBJECT
337. PICK-OBJECT-FROM-FILE
338. FORGET-CURRENT-FILE
339. PICK-A-FILE

340. PUT-OBJECT-IN-FILE

        WORKING MEMORY
((NUMBER-OF-PIECES 24)
 (NUMBER-OF-FILES 2)
 (IN-PUZZLE (PIECE 64 BROWN))
 (IN-PUZZLE (PIECE 63 BROWN))
 (IN-PUZZLE (PIECE 62 BROWN))
 (IN-PUZZLE (PIECE 52 BROWN))
 (IN-PUZZLE (PIECE 42 BLACK))
 (IN-PUZZLE (PIECE 32 BLACK))
 (IN-PUZZLE (PIECE 22 RED))
 (IN-PUZZLE (PIECE 65 BLACK))
 (IN-PUZZLE (PIECE 55 BLACK))
 (IN-PUZZLE (PIECE 12 BLACK))
 (IN-PUZZLE (PIECE 13 BLACK))
 (IN-PUZZLE (PIECE 14 BLACK))
 (IN-PUZZLE (PIECE 15 BLACK))
 (IN-PUZZLE (PIECE 25 BLACK))
 (IN-PUZZLE (PIECE 35 SILVER))
```

```
(IN-PUZZLE (PIECE 45 BLACK))
(IN-PUZZLE (PIECE 53 BROWN))
(IN-PUZZLE (PIECE 54 BROWN))
(CURRENT-COLOR SILVER)
(HOLDING NOTHING)
(LOOKING-AT (PIECE 44 SILVER))
(CURRENT-FILE 2)
(REMEMBERED-FILE 1)
(REMEMBERED-OBJECT NONE)
(ALL-FILES 2 1)
(FILE 1 (PIECE 43 BLACK) (PIECE 33 RED)
        (PIECE 24 RED) (PIECE 23 BLACK))
(FILE 2 (PIECE 34 SILVER) (PIECE 44 SILVER))
(NUMBER-IN-PUZZLE 18)
(JOINED ((L 64 107 BROWN) (R 63 -107 BROWN))
        ((L 63 -106 BROWN) (R 62 106 BROWN))
        ((T 62 -101 BROWN) (B 52 101 BROWN))
        ((T 52 -69 BLACK) (B 42 69 BLACK))
        ((T 42 53 BLACK) (B 32 -53 BLACK))
        ((T 32 -35 BLACK) (B 22 35 RED))
        ((R 64 -108 BROWN) (L 65 108 BROWN))
        ((T 65 104 BLACK) (B 55 -104 BLACK))
        ((T 22 27 BLACK) (B 12 -27 BLACK))
        ((R 12 23 BLACK) (L 13 -23 BLACK))
        ((R 13 -24 BLACK) (L 14 24 BLACK))
        ((R 14 31 BLACK) (L 15 -31 BLACK))
        ((B 15 38 BLACK) (T 25 -38 BLACK))
        ((B 25 47 BLACK) (T 35 -47 SILVER))
        ((B 35 60 SILVER) (T 45 -60 BLACK))
        ((R 52 -86 BROWN) (L 53 86 BROWN))
        ((T 63 102 BROWN) (B 53 -102 BROWN))
        ((R 53 87 BROWN) (L 54 -87 BROWN))
        ((T 64 -103 BROWN) (B 54 103 BROWN))
        ((L 55 88 BLACK) (R 54 -88 BROWN))))
    341. LOOK-AT-FIRST-IN-FILE
    342. PICK-OBJECT-FROM-FILE
    343. FIT-PIECE-IN-PUZZLE
    344. PIECE-PUT-IN-PUZZLE
    345. LOOK-AT-FIRST-IN-FILE
    346. PICK-OBJECT-FROM-FILE
    347. FIT-PIECE-IN-PUZZLE
    348. PIECE-PUT-IN-PUZZLE
    349. FORGET-CURRENT-FILE

    350. PICK-A-FILE
    351. LOOK-AT-FIRST-IN-FILE
    352. REMEMBER-CURRENT-OBJECT
    353. FIND-COLOR-OF-PIECE
    354. LOOK-AT-NEXT-IN-FILE
    355. LOOK-AT-NEXT-IN-FILE
    356. LOOK-AT-NEXT-IN-FILE
    357. PICK-OBJECT-FROM-FILE
    358. FORGET-CURRENT-FILE
    359. PICK-A-FILE

    360. PUT-OBJECT-IN-FILE
    361. FORGET-CURRENT-FILE
    362. PICK-A-FILE
    363. LOOK-AT-FIRST-IN-FILE
    364. FORGET-REMEMBERED-OBJECT
```

```
365. PICK-OBJECT-FROM-FILE
366. FORGET-CURRENT-FILE
367. PICK-A-FILE
368. PUT-OBJECT-IN-FILE
369. LOOK-AT-FIRST-IN-FILE

370. PICK-OBJECT-FROM-FILE
371. FIT-PIECE-IN-PUZZLE
372. PIECE-PUT-IN-PUZZLE
373. LOOK-AT-FIRST-IN-FILE
374. PICK-OBJECT-FROM-FILE
375. FIT-PIECE-IN-PUZZLE
376. PIECE-PUT-IN-PUZZLE
377. FORGET-CURRENT-FILE
378. PICK-A-FILE
379. LOOK-AT-FIRST-IN-FILE

380. REMEMBER-CURRENT-OBJECT
381. FIND-COLOR-OF-PIECE
382. LOOK-AT-NEXT-IN-FILE
383. PICK-OBJECT-FROM-FILE
384. FORGET-CURRENT-FILE
385. PICK-A-FILE
386. PUT-OBJECT-IN-FILE
387. FORGET-CURRENT-FILE
388. PICK-A-FILE
389. LOOK-AT-FIRST-IN-FILE

390. FORGET-REMEMBERED-OBJECT
391. PICK-OBJECT-FROM-FILE
392. FORGET-CURRENT-FILE
393. PICK-A-FILE
394. PUT-OBJECT-IN-FILE
395. LOOK-AT-FIRST-IN-FILE
396. PICK-OBJECT-FROM-FILE
397. FIT-PIECE-IN-PUZZLE
398. PIECE-PUT-IN-PUZZLE
399. LOOK-AT-FIRST-IN-FILE

400. PICK-OBJECT-FROM-FILE
401. FIT-PIECE-IN-PUZZLE

        WORKING MEMORY
((NUMBER-OF-PIECES 24)
 (NUMBER-OF-FILES 2)
 (IN-PUZZLE (PIECE 64 BROWN))
 (IN-PUZZLE (PIECE 63 BROWN))
 (IN-PUZZLE (PIECE 62 BROWN))
 (IN-PUZZLE (PIECE 52 BROWN))
 (IN-PUZZLE (PIECE 42 BLACK))
 (IN-PUZZLE (PIECE 32 BLACK))
 (IN-PUZZLE (PIECE 22 RED))
 (IN-PUZZLE (PIECE 65 BLACK))
 (IN-PUZZLE (PIECE 55 BLACK))
 (IN-PUZZLE (PIECE 12 BLACK))
 (IN-PUZZLE (PIECE 13 BLACK))
 (IN-PUZZLE (PIECE 14 BLACK))
 (IN-PUZZLE (PIECE 15 BLACK))
 (IN-PUZZLE (PIECE 25 BLACK))
 (IN-PUZZLE (PIECE 35 SILVER))
```

```
(IN-PUZZLE (PIECE 45 BLACK))
(IN-PUZZLE (PIECE 53 BROWN))
(IN-PUZZLE (PIECE 54 BROWN))
(IN-PUZZLE (PIECE 34 SILVER))
(IN-PUZZLE (PIECE 44 SILVER))
(IN-PUZZLE (PIECE 23 BLACK))
(IN-PUZZLE (PIECE 43 BLACK))
(IN-PUZZLE (PIECE 24 RED))
(CURRENT-COLOR RED)
(HOLDING (PIECE 33 RED))
(LOOKING-AT (PIECE 33 RED))
(CURRENT-FILE 2)
(REMEMBERED-FILE 1)
(REMEMBERED-OBJECT NONE)
(ALL-FILES 2 1)
(FILE 1)
(PILE 2)
(NUMBER-IN-PUZZLE 23)
(JOINED ((L 64 107 BROWN) (R 63 -107 BROWN))
        ((L 63 -106 BROWN) (R 62 106 BROWN))
        ((T 62 -101 BROWN) (B 52 101 BROWN))
        ((T 52 -69 BLACK) (B 42 69 BLACK))
        ((T 42 53 BLACK) (B 32 -53 BLACK))
        ((T 32 -35 BLACK) (B 22 35 RED))
        ((R 64 -108 BROWN) (L 65 108 BROWN))
        ((T 65 104 BLACK) (B 55 -104 BLACK))
        ((T 22 27 BLACK) (B 12 -27 BLACK))
        ((R 12 23 BLACK) (L 13 -23 BLACK))
        ((R 13 -24 BLACK) (L 14 24 BLACK))
        ((R 14 31 BLACK) (L 15 -31 BLACK))
        ((B 15 38 BLACK) (T 25 -38 BLACK))
        ((B 25 47 BLACK) (T 35 -47 SILVER))
        ((B 35 60 SILVER) (T 45 -60 BLACK))
        ((R 52 -86 BROWN) (L 53 86 BROWN))
        ((T 63 102 BROWN) (B 53 -102 BROWN))
        ((R 53 87 BROWN) (L 54 -87 BROWN))
        ((T 64 -103 BROWN) (B 54 103 BROWN))
        ((L 55 88 BLACK) (R 54 -88 BROWN))
        ((L 35 -56 SILVER) (R 34 56 SILVER))
        ((B 34 -57 SILVER) (T 44 57 SILVER))
        ((T 54 77 SILVER) (B 44 -77 SILVER))
        ((L 45 -76 BLACK) (R 44 76 SILVER))
        ((B 13 -30 BLACK) (T 23 30 BLACK))
        ((R 22 -29 RED) (L 23 29 BLACK))
        ((R 42 -68 BLACK) (L 43 68 BLACK))
        ((B 14 -42 BLACK) (T 24 42 BLACK))
        ((R 23 -36 BLACK) (L 24 36 RED))
        ((T 34 -46 BLACK) (B 24 46 RED))
        ((L 25 44 BLACK) (R 24 -44 BLACK))))

    402. PIECE-PUT-IN-PUZZLE
    403. FORGET-CURRENT-PILE
    404. PICK-A-FILE
    405. DESTROY-A-PILE
    406. PICK-A-FILE
    407. DESTROY-A-PILE
    408. PUZZLE-IS-FINISHED

(END -- EXPLICIT HALT)
(1 -- META RULE CALLS)
```

```
33 PRODUCTIONS IN SYSTEM
127/300 NODES
408 PRODUCTIONS-FIRED
3642 PRODUCTIONS-INSTANTIATED
1831 WM-TRANSACTIONS(68762 NODE ACTIVATIONS)
(68035 TESTS PERFORMED)
(131 MAXIMUM WM SIZE)
(101.62990 MEAN WM SIZE)
(33 MAXIMUM CS SIZE)
(12.96324 MEAN CS SIZE)
(177 MAXIMUM NUMBER TOKENS STORED)
(117.18382 MEAN NUMBER TOKENS STORED)

11.5 MINUTES CPU
```

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER # 18	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Automatic Discovery of Heuristics for Non- deterministic Programs from Sample Execution Traces.		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s)		8. CONTRACT OR GRANT NUMBER(s) ONR: N00014-75-C-0571
9. PERFORMING ORGANIZATION NAME AND ADDRESS Courant Institute of Mathematical Sciences New York University 251 Mercer Street, New York, N.Y. 10012		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research		12. REPORT DATE September 1979
		13. NUMBER OF PAGES 168
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Department of the Navy Arlington, Virginia 22217		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) None		
18. SUPPLEMENTARY NOTES None		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Heuristics, Nondeterministic program, Pattern Recognition, Production System, Trace Sequence		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) See attached.		

20.

ABSTRACT: During the last few years a number of relatively
effective (AI) programs have been written incorporating
considerable amounts of problem specific knowledge. Conse-
quently, the problem of encoding such knowledge in a useful
form has emerged as one of the central problems of AI. In
particular, declarative representations have attracted much
attention partly because of the relative ease with which
knowledge can be communicated in this form. Unfortunately,
straightforward implementation of declaratively specified
knowledge corresponds to a nondeterministic program which
incurs enormous computational costs.

This thesis investigates one way to limit this cost.
The approach we take is to develop control heuristics for a
family of problems from traces of sample solutions generated
during a training session with a human expert. Algorithms
are presented which recognize a set of patterns in the
sequence of 'knowledge applications' and which compile
descriptions of these patterns in a control language, called
CRAPS. More specifically, patterns of repeating, parallel
and common sequences are considered in the analysis. The
analysis also produces a set of meta-rules which aid the CRAPS
description in the event the sequencing it specifies is
inappropriate. The CRAPS description and meta-rules are
then used for guidance in solving subsequent problems.

The below listing is the revised official distribution list for the technical, annual, and final reports for Contract N00014-75-C-0571. This list supersedes the one dated 27 March 1974.

Defense Documentation Center Cameron Station Alexandria, Virginia 22314	12 copies
Office of Naval Research Information Systems Program Code 437 Arlington, Virginia 22217	2 copies
Office of Naval Research Code 102IP Arlington, Virginia 22217	6 copies
Office of Naval Research Branch Office, Boston 495 Summer Street Boston, Massachusetts 02210	1 copy
Office of Naval Research Branch Office, Chicago 536 South Clark Street Chicago, Illinois 60605	1 copy
Office of Naval Research Branch Office, Pasadena 1030 East Green Street Pasadena, California 91106	1 copy
New York Area Office 715 Broadway - 5th Floor New York, New York 10003	1 copy

Naval Research Laboratory   6 copies
Technical Information Division, Code 2627
Washington, D.C. 20375

Dr. A. L. Slafkosky   1 copy
Scientific Advisor
Commandant of the Marine Corps (Code RD-1)
Washington, D.C. 20380

Office of Naval Research   1 copy
Code 455
Arlington, Virginia 22217

Office of Naval Research   1 copy
Code 458
Arlington, Virginia 22217

Naval Electronics Laboratory Center   1 copy
Advanced Software Technology Division
Code 5200
San Diego, California 92152

Mr. E. H. Gleissner   1 copy
Naval Ship Research & Development Center
Computation and Mathematics Department
Bethesda, Maryland 20084

Captain Grace M. Hopper   1 copy
NAICOM/MIS Planning Branch (OP-916D)
Office of Chief of Naval Operations
Washington, D.C. 20350

Mr. Kin B. Thompson   1 copy
Technical Director
Information Systems Division (OP-91T)
Office of Chief of Naval Operations
Washington, D.C. 20350

Dr. R. T. Chien   1 copy
University of Illinois
Coordinated Science Laboratory
Urbana, Illinois 61801

Director, National Security Agency   1 copy
ATTN: R53, Mr. Glick
Fort G.G. Meade, Maryland 30755

Office of Naval Research   1 copy
Code 432
Arlington, Virginia 22217

Professor John Birk   1 copy
University of Rhode Island
Dept. of Electrical Engineering
Kelley Hall
Kingston, Rhode Island 02881